

Table of Contents (IN3020&4020, 2024)

Lecture	Part	Topic	Slide
1	– Intro & recap	Introduction and motivation	2
2	– Intro & recap	Relational model	33
3	– Intro & recap	SQL summary - 1	76
4	– Intro & recap	SQL summary - 2	108
5, 6	1: Query processing	Relational algebra	146
7, 8	1: Query processing	Relational algebra for SQL	198
9, 10	1: Query processing	Indexes	234
11, 12	1: Query processing	Query compilation and optimisation	291
13–15	1: Query processing	Query evaluation	346
16–18	2: Transaction management	Transaction processing concepts	405
19–21	2: Transaction management	Concurrency control techniques	464
22, 23	2: Transaction management	Recovery techniques	538
24	–	Database security	594
25	3: NoSQL	Overview	631
26–28	3: NoSQL	Graph databases	667
29	–	DBMS for Big Data: Challenges	765

IN3020&4020 – Database Systems (2024)

Part: Intro & Recap

Lecture 1: Introduction and Motivation

15 January

Egor V. Kostylev and **Leif Harald Karlsen**

IFI, University of Oslo

egork@ifi.uio.no



About us: Egor V. Kostylev

2004–2009: PhD, Lomonosov Moscow State University
Programming Languages

2010–2013: PostDoc, University of Edinburgh
Databases

2013–2020: Lecturer, University of Oxford
+ Knowledge Representation

from **Sep. 2020**: Associate Professor, University of Oslo
+ Hybrid AI

Language: Jeg snakker (litt) **norsk**, but the course is in **English**

How to find me: egork@ifi.uio.no, office 8165 in OJD

2014–2018: PhD, University of Oslo

Spatial Databases

2018–2019: Head Engineer, SIRIUS/University of Oslo

Semantic Technologies

2018: Lecturer, Norwegian Business School

Databases

from **Aug. 2019:** Senior Lecturer, University of Oslo

Databases and Data Engineering

How to find me: leifhka@ifi.uio.no, office 9168 in OJD

Physical:

- main part of the course, usually 2 a week (other parts described below)
- an opportunity for a dialogue: interrupt us and ask questions
- we will ask questions sometimes, on the fly or using **Menti**

Lectures will be **recorded**:

- videos will be **published** (very) soon after the lecture
- no promise!
- do not forget about **mandatories** (see below):
you need to follow the course during the term to succeed

What this is about hierarchy

IN2090 – Databaser og datamodellering (Leif Harald Karlsen)

- **what** databases are and **how to use** them
- pre-requisite (obligatoriske forkunnskaper) for **IN3020**
- not for **IN4020** (knowledge of the material is expected, **but ...**)

IN3020&4020 – Database systems

- **how** databases **work** inside
- pre-requisite for **IN5040**
- **NOT an advanced SQL course!**

IN5040 – Advanced database systems for Big Data (Vera Hermine Goebel)

- **new challenges** in data management

Our *general strategy* for IN2090 material:

- Not important for us: ignored
- We build upon: presented, but quickly
- We study deeper: **repeated in detail** (and extended)

Our use of IN2090 synopsis:

1. **Data modelling**

- **relational model** (including keys)
- **relational algebra**
- entity-relation (ER) diagrams
- normal forms and decomposition

2. **SQL**

- syntax and semantics of Data Query Sub-Language
- **security in DBs**
- **indexing and query processing**

Databases and Database Management Systems

Database:

- collection of related **data**—that is, known facts with implicit meaning—organised in some way (**data model**)
- *In this course*: **relational databases** (mostly), **RDF knowledge graphs**, etc.

Database management system (DBMS):

- a **software** that facilitates **defining, constructing, manipulating,** and **sharing** databases by various users and applications
- *In this course*: **SQL engines** (mostly), **SPARQL engines**, etc.

We often use ‘database’ and ‘DBMS’ **interchangeably**, as well as **omit** ‘relational’ (when this does not cause confusion, which is usually the case)

Why to bother about databases?

CSV + Python:

```
import csv
import os
filea="a.csv"
fileb="b.csv"
temp="temp.csv"
src1=csv.reader(open(filea,"r"),delimiter=",")
src2=csv.reader(open(fileb,"r"),delimiter=",")
src2_dict={}
for row in src2:
    src2_dict[row[0]] = row[1]
with open(temp,"w") as fout:
    csvwriter=csv.writer(fout, delimiter=delim)
    for row in src1:
        if row[1] in src2_dict:
            row[3]=src2_dict[row[1]]
            csvwriter.writerow(row)
os.rename(temp, filea)
```

Database + SQL:

```
UPDATE a
SET c4=b.c2
FROM b
WHERE a.c2=b.c1;
```

Question: Why the right is better than the left?

Why to bother about (dedicated) databases?

CSV + Python:

```
...  
for row in src2:  
    src2_dict[row[0]] = row[1]  
...
```

Database + SQL:

```
UPDATE a  
SET c4=b.c2  
...
```

Question: Why the right is better than the left?

Answer 1: SQL is easier to learn than Python

Answer 2: SQL is **declarative**:

- we tell the engine **what we want** rather than **how to compute it**
- **easier** (and shorter) to formulate
- much more space for **optimisation**:
 - rewrite query into a more efficient one
 - store data in an easy-to-access way
 - efficient treatment of many users with complex transactions

Why we concentrate on relational databases?

Relational databases (with SQL):

- are the **most popular** database architecture
- have good **balance** between simplicity and efficiency
- **natural** and **intuitive** to work with

'Databases' usually mean 'relational databases'

For us, relational databases are just a **typical example**, and many challenges and approaches we will study are applicable to other database architectures

In the end of the course, we will look at other, **NoSQL** architectures:

- RDF + SPARQL, Neo4j + Cypher, etc.

Inside the database 'black box'

- What is apparent to us is that we **write** an SQL query, and the database **does something** and comes up with an answer
- But there is a lot that happens **behind the scenes**, inside the 'black box'
- **This course** will show what happens in the 'black box', so that we understand the common **challenges** and how to deal with them

Challenge 1: Query optimisation (example)

SELECT B, C, Y

FROM R, S

WHERE W = X AND A = 3 AND Z = 'a'

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

~>

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	2	f	c
2	c	6	...	2	1	a	a
...	2	f	c
3	r	8	...	7	1	a	a
...	2	f	c
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	2	f	c
...	v
2	j	0	...	3	1	a	a
...	2	c	c
...	v
3	t	5	...	9	1	a	a
...	2	f	c
...	v
7	e	3	...	3	1	a	a
...	2	f	c
...	v

~>

B	C	Y
r	8	k

Challenge 1: Query optimisation (example)

```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z = 'a'
```

Naive approach:

cross each tuple in **R** with each in **S**; select using condition; project

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

~>

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	2	f	c
2	c	6	...	2	1	a	a
...	2	f	c
3	r	8	...	7	1	a	a
...	2	f	c
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	2	f	c
...	v
2	i	0	...	3	1	a	a
...	2	c	c
3	t	5	...	9	1	a	a
...	2	f	c
7	e	3	...	3	1	a	a
...	2	f	c
...	v

~>

B	C	Y
r	8	k

Challenge 1: Query optimisation (example)

```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z = 'a'
```

Question: Can we do it smarter?

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

↔

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	2	f	c
2	c	6	...	2	1	a	a
...	2	f	c
3	r	8	...	7	1	a	a
...	2	f	c
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	2	f	c
...	v
2	j	0	...	3	1	a	a
...	2	c	c
...	v
3	t	5	...	9	1	a	a
...	2	f	c
...	v
7	e	3	...	3	1	a	a
...	2	f	c
...	v

↔

B	C	Y
r	8	k

Challenge 1: Query optimisation (example)

```
SELECT B, C, Y
FROM R, S
WHERE W = X AND A = 3 AND Z = 'a'
```

Smarter approach (much smaller tables to manipulate):
select relevant tuples in **R** and **S**; cross with selecting; project

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c



Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c



A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a



B	C	Y
r	8	k

Challenge 2: Efficient data storage (example)

```
SELECT *  
FROM R  
WHERE A = 3
```

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Naive approach: Try each tuple one by one, select relevant

Question: Can we represent data smarter so trying all is usually not necessary?

A solution: Sort tuples in the table in advance according to **A**, use binary search when evaluating the query

Problem: Works only for **A**. What if we also expect queries for **W**?

Better solution: Use *indexes!* (See the course)

Challenge 3: Efficient transaction management

Instructions over databases come in sequences, called **transactions**

- 'instruction' may be a **query** (update, select, etc.) or a finer **piece of a query** (e.g., read the tuple by this pointer, add this tuple at the end of this table, etc.)
- each transaction is expected to be evaluated **in full** ('all or nothing') to preserve consistency
- transactions may come in parallel (e.g., from different users)

Requirements can be formalised as **ACID** principles:

Atomicity, Consistency, Isolation, Durability

Approach:

Transaction Management (locking, logging, buffering, etc.)

How DBMS work inside (relational and NoSQL):

- How queries are optimised and evaluated
- How data is stored and managed
- ACID principles
- Transaction management types (i.e., isolation levels)
- Theoretical and practical aspects of database security

Three main parts:(different sizes!):

- Intro and SQL recap **Egor**
- 1. **Query processing in relational databases** **Egor**
 - relational algebra, indexes, query optimisation and evaluation

- 2. **Transaction management in relational databases** **Leif Harald**
 - ACID, serialisation, concurrency control, isolation levels
(+ DB security)
- Advanced DBMS Architecture (for IN4020) **Leif Harald**
- 3. **NoSQL DBMS** **Leif Harald**
 - knowledge graphs, column-based, etc.
- A bit about emerging technologies and research **Leif Harald**

Lecture schedule – 1 (preliminary, with rough borders)

#	Date	Part	Topic
1	15.01 (Mon)	Intro+SQL	Introduction
2	16.01 (Tue)	Intro+SQL	Relational Model
3	22.01 (Mon)	Intro+SQL	SQL recap – 1
4	23.01 (Tue)	Intro+SQL	SQL recap – 2
5	29.01 (Mon)	Part 1: Query Processing	Relational Algebra – 1
6	30.01 (Tue)	Part 1: Query Processing	Relational Algebra – 2
7	05.02 (Mon)	Part 1: Query Processing	Relational Algebra – 3
8	06.02 (Tue)	Part 1: Query Processing	Relational Algebra – 4
9	12.02 (Mon)	Part 1: Query Processing	Indexing – 1
10	13.02 (Tue)	Part 1: Query Processing	Indexing – 2
11	19.02 (Mon)	Part 1: Query Processing	Query compilation – 1
12	20.02 (Tue)	Part 1: Query Processing	Query compilation – 2
13	27.02 (Tue)	Part 1: Query Processing	Query evaluation – 1
14	04.03 (Mon)	Part 1: Query Processing	Query evaluation – 2

Mondays: 14:15–16:00 (OJD, Smalltalk)

Tuesdays: 10:15–12:00 (KN, Lille Aud.)

Lecture schedule – 2 (preliminary, with rough borders)

#	Date	Part	Topic
15	11.03 (Mon)	Part 2: TX Management	Transaction Concepts – 1
16	12.03 (Tue)	Part 2: TX Management	Transaction Concepts – 2
17	18.03 (Mon)	Part 2: TX Management	Transaction Concepts – 3
18	19.03 (Tue)	Part 2: TX Management	Concurrency Control – 1
19	02.04 (Tue)	Part 2: TX Management	Concurrency Control – 2
20	08.04 (Mon)	Part 2: TX Management	Concurrency Control – 3
21	09.04 (Tue)	Part 2: TX Management	Recovery Protocols – 1
22	15.04 (Mon)	Part 2: TX Management	Recovery Protocols – 2
23	16.04 (Tue)	(Part 2+)	Intro do DB Security
24	22.04 (Mon)	Part 3: NoSQL	NoSQL Overview
25	23.04 (Tue)	Part 3: NoSQL	Graph Databases – 1
26	29.04 (Mon)	Part 3: NoSQL	Graph Databases – 2
27	30.04 (Tue)	Part 3: NoSQL	Graph Databases – 3
28	06.05 (Mon)	(no part, for IN4020)	DBMS for Big Data
29	07.05 (Tue)	Summary and Exam	Summary
30	13.05 (Mon)	Summary and Exam	2023 Exam analysis

Mondays: 14:15–16:00, **Tuesdays:** 10:15–12:00

Main materials: Lecture slides and videos

- slides will be published (shortly) **before** the lecture at the Web page of the course:

<https://www.uio.no/studier/emner/matnat/ifi/IN3020/v24/timeplan/index.html>

- slides may have typos: please, let us know (egork@ifi.uio.no) if you find bugs
- videos will be published (shortly) **after** the lectures at the page

Secondary materials: Group and mandatory exercises (+ 2022&23 exams, see below)

Other materials:

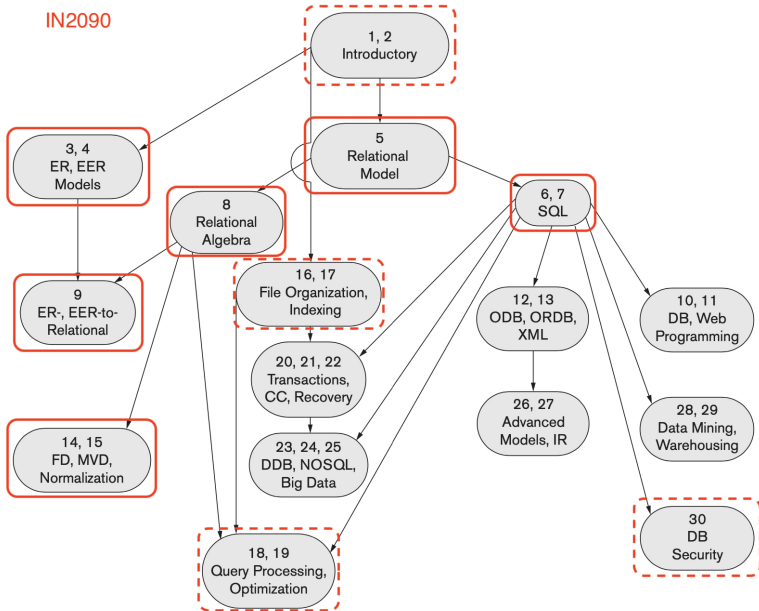
1. *Fundamentals of Database Systems*
by R. Elmasri and S.B. Navathe (7th ed.)
2. *Database systems – the complete book*
by H. Garcia-Molina, J.D. Ullman, and J. Widom (2nd ed.)
 - only some **parts** are relevant
 - **not everything** is covered
 - many things are presented **differently**
3. Other books and resources:
 - *Foundations of Databases* by Abiteboul, Hull, and Vianu
<http://webdam.inria.fr/Alice/> (official)
 - **Wikipedia**, **PostgreSQL documentation**, etc.



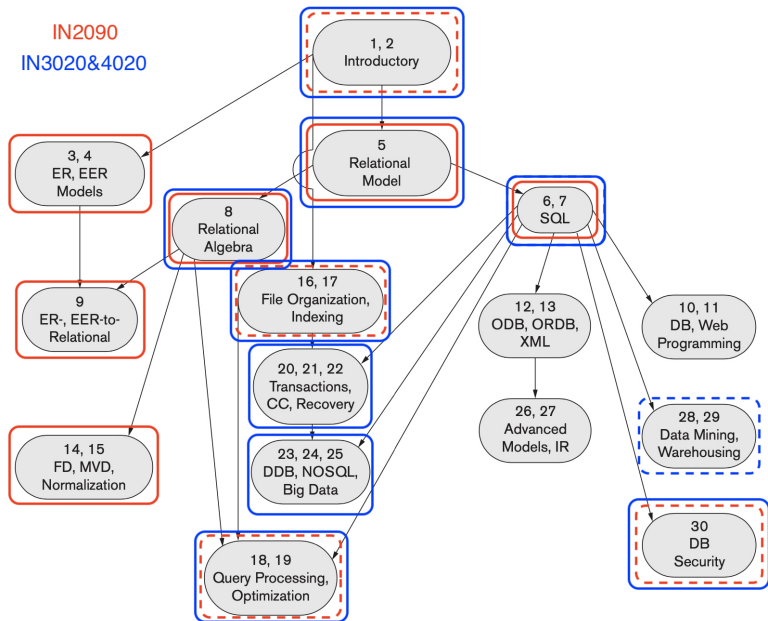
I will start **every lecture** with a list of relevant materials

Elmasri&Navathe chapter map (p. 13)

IN2090



Elmasri&Navathe chapter map (p. 13)



Group sessions

We have 3 groups with weekly sessions (each week except this)

- Group 1: Tue 12:15–14:00, Seminarrom Sed
Teacher: *Md Mahamodul Islam* (mdmi@ifi.uio.no)
- Group 2: Fri 10:15–12:00, Datastue Fortress
Teacher: *Mohammad Mainul Hasan* (mohah@ifi.uio.no)
- Group 3: Mon (+ 1 week!) 10:15–12:00, Datastue Fortress
Teacher: *Erik Snilsberg* (eriksni@ifi.uio.no)

Discuss problems (published on previous Monday each week)

- it is desirable that you have a look at them in advance
- roughly covers the previous week material
- a more difficult problem (with a 'star', similar to mandatories and exam)
- solutions will be published (after the Friday's session)

May include some demonstrations with PostgreSQL (see below)

Also discuss completed mandatories (see below)

Mandatory exercises

We have 3 mandatories:

#	Topic	Deadline
1	Relational Algebra	??
2	Query Processing	??
3	Transaction Management	??

- Published **2 weeks** before the deadline
- Submitted in **Devilry**
- Check the **rules** for the mandatory exercises (obligement)
- All **necessary** for the exam
- **Extensions** are possible if properly justified (illness, etc.)
- Marked within two weeks (or quicker) by group teachers (**pass/not pass**)
- May be **resubmitted** if not passed
- Deadlines may **be changed** (will let you know in advance)
- **Short**

Exam formal details:

- **Time:** 30 May 2024, 9am **Duration:** 4 hours
- **Place:** Silurveien 2 Sal 3B **System:** Inspera
- **Withdrawal deadline:** 1 May

Exam rules and contents:

- no **collaboration** (standard plagiarism rules)
- lecture slides are **available** during the exam
- **no other materials** are available
- you should be able to **apply** lecture materials
- if you are comfortable with **mandatories**, you should be fine with the exam
- **22&23 exams** (but not the before) is good training material
(do **not** recommend to look at 23 before final training)

Communication (we need extra effort)

Main entry point: the Web page of the course:

<https://www.uio.no/studier/emner/matnat/ifi/IN3020/v24/index.html>

- **schedule** of lectures and group sessions
- relevant **links** slides, videos, group exercises
- **messages** (changes, mandatory sheets, links to other relevant systems)

Discussions: Astro Discourse

<https://astro-discourse.uio.no/c/in3020-24v> (updated link!)

- approved by UiO, have good reputation from other lecturers
- student instructions can be found on Web page of the course
- We should **respond** within 1 working day

Lectures: I stay for some time after each lecture

Q1: Is this course theoretical or practical?

A1: **Both (or neither)**: no programming is involved, but we discuss how real software works (cf. *Compilers* course)

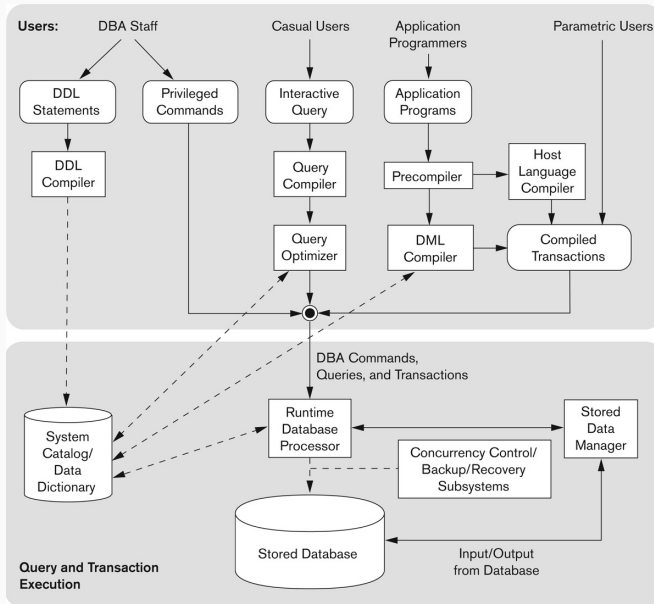
Q2: Do I need to know how to use SQL systems (PostgreSQL)?

A2: **Not necessary, but expected**: the mandatories and exam are 'on-paper', but we will encourage you to experiment time to time and demonstrate how to do this

Q3: How different is the course from the last year?

A3: **Quite similar**: you may look at slides and videos, but nothing is promised (I may change and adapt some parts)

Example DBMS architecture



IN3020&4020 – Database Systems (2024)

Part: Intro & Recap

Lecture 2: Relational Model

16 January

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Physical:

- main part of the course, 2 a week
- an opportunity for a dialogue: interrupt me and ask questions
- I will ask questions sometimes

Lectures will be **recorded**:

- videos will be **published** (very) soon after the lecture
- do not forget about **mandatories**:
you need to follow the course during the term to succeed

Materials to read

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th ed.): Chapter 5, Sections 5.1&5.2
2. *Database systems – the complete book* by H. Garcia-Molina, J.D. Ullman, and J. Widom (2nd ed.): Section 2.1, 2.2, 2.5 (+ 3.1, 7.1, 7.2)
3. *Foundations of Databases* by S. Abiteboul, R. Hull, and V. Vianu, Chapter 3
<http://webdam.inria.fr/Alice/>
4. [IN2090 – Databaser og datamodellering](#) (Leif Harald Karlsen), Lecture 3
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
5. [Wikipedia](#), etc.

There are slight **variations** (we will discuss)

1. History of data models
2. Classic relational model (a version of)
3. Integrity constraints (keys and foreign keys)

1. History of data models

Database: collection of related **data**—that is, known facts with implicit meaning—organised in some way

Data model: the conceptual representation of data usually, **abstract** representation

- **concentrates** on details important to usual database users
- **hides** the storage and implementation details that are not important to these users

Brief history (look them up)

Early days (1960's) of digital data management:

- **Mainframe period**, 1 'big' machine for (that-time) 1M\$
- 512 Kbyte RAM, 50 Mbyte disk
- Other I/O-equipment magnetic & paper tape, punched-card reader and teletype (thus, unix has funny abbreviations such as `tty` for the «terminal»)

- **hierarchical** and **network** data models
- **ad-hoc** own query languages tailored for specific applications, largely **procedural** (i.e., not **declarative**)

Hierarchical database (actually, data) model

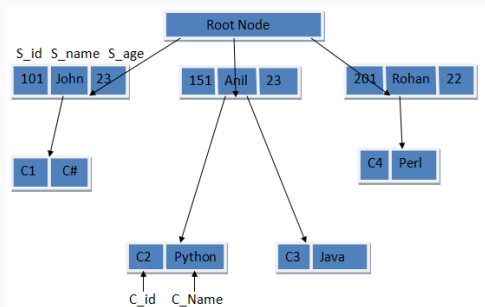
Underlies IBM's **Information Management System (IMS)** DBMS

Used in **bill of materials** for **Apollo** space program

Data is represented in a **tree-like** structure

Edges are **parent-child** relationships

Own **query language** (called **DL/I**, tightly integrated into **COBOL**)



Network database model

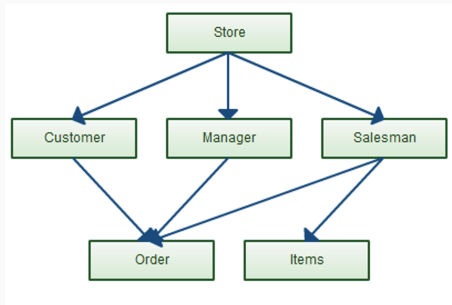
Underlies General Electric's **Integrated Data Store (IDS)** DBMS

Used by BT **Customer Service System** (and other projects)

Generalised hierarchical model: several parents of a child are allowed

Performance-oriented with low level of abstraction

Own (difficult to learn) **query language**



E. F. Codd, 1970: *A relational model of data for large shared data banks* (Commun. ACM 13, 6):

"Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed."

What did Codd mean?

User **should not** see the internal representation (files, pointers):

- truly **abstract** data model

Relations as an abstract data structure:

- simple **universal** data model

Has a **declarative** query language of **Relational Algebra** expressions:

- has **formal** theoretical foundations
- **simple** and concise
- allows for **efficient** implementations: a lot of space for optimisation

Since then ‘**databases**’ usually mean ‘**relational databases**’:

- **SQL** is based on (variants of) relational model and algebra

Before we proceed to the details...

Since 2000 back to roots:

- XML is essentially a **hierarchical model**
- Knowledge Graphs (e.g., RDF) are essentially a **network model**

Question: Why they lost in the 70s, but regain popularity now?

Answer: Just *fancy*.

Before we proceed to the details...

Since 2000 back to roots:

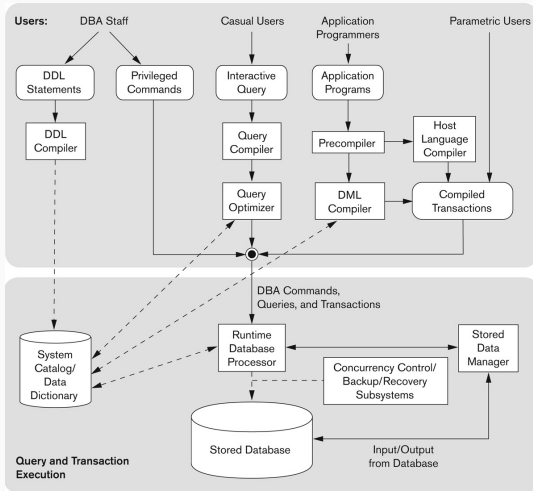
- XML is essentially a **hierarchical model**
- Knowledge Graphs (e.g., RDF) are essentially a **network model**

Question: Why they lost in the 70s, but regain popularity now?

Real answer: we are on a **different level**:

- we **understand much better** what different data models (and query languages) are good for
- these models have **universal** and **declarative** query languages (XQuery, SPARQL, Cypher, etc.)
- the languages are much more **efficient** than the ones of 60's in all components

The place of the relation model in DBMS architecture



Abstract model: this is what all the **users interact** with, but *not* how data is **stored in reality**

2. Classical relational model

Relations: intuition and terminology

A relation can be seen as a **table of values**. Something like this:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Relations: intuition and terminology

A relation can be seen as a **table of values**. Or this:

Tutorials

ID	site	tutorial	topic
1	w3schools	SQL_2003STD	Database
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Database
5	w3resource	MySQL	Database

Relations: intuition and terminology

A relation can be seen as a **table of values**. Something like this:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

- **Tutorials:** **relation name**
- ID, site, tutorial, topic: **attribute names** (or just **attributes**), collectively: **signature**
- Relation name + signature: **relation schema** (or just **relation**)
- Rows: **tuples, records, or instances**
collectively: **relation state**
- Elements of tuples: **attribute values** (or just **attributes** as well)

Relation

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

- Each attribute has a **domain** (i.e., a set of potential values): integers, strings, etc. (often not written explicitly)
- Different attributes may have the same domain
- Attribute values **are from** the associated domain

Relation

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

- Tuples are **unordered** and **unique** (i.e., not identical)

Same relation

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
1	w3schools	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

- Tuples are **unordered** and **unique** (i.e., not identical)

Not a relation(!)

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
1	w3schools	SQL_2003STD	Databases
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases

- Tuples are **unordered** and **unique** (i.e., not identical)

Also same relation

Tutorials:

<i>site</i>	<i>ID</i>	<i>tutorial</i>	<i>topic</i>
w3schools	1	SQL_2003STD	Databases
w3schools	2	HTML_5	WebDev
w3schools	3	CSS_3	WebDev
w3resource	4	SQL_2003STD	Databases
w3resource	5	MySQL	Databases

- Tuples are **unordered** and **unique** (i.e., not identical)
- Attributes are also **unordered** and **unique**
(not so essential, but convenient for us;
may be different in different formalisations)

Formally: relation schema

Domain: a set of values

- may be **finite** (e.g., countries {*Norway*, *UK*, *India*, ...}) or **infinite** (e.g., integers)
- **atomic**—that is, values have no essential internal structure (e.g., not sets or tuples themselves)
- usually, has a corresponding **datatype** in the query language

Attribute (i.e., ‘column name’): a string with an associated domain

- domain of attribute A is written $dom(A)$
- for example, $dom(ID)$ is natural numbers

Relation schema: a relation name R (just string) and several attributes A_1, A_2, \dots, A_n

- written as $R[A_1, A_2, \dots, A_n]$ (or $R[A_2, A_1, \dots, A_n]$, order is irrelevant)
- n is called the **arity** of R (**question:** can the arity be 0?)
- for example $Tutorials[ID, site, tutorial, topic]$ has arity 4

Formally: tuples

Tuple (or **record**, **row**, **instance**) over schema $R[A_1, \dots, A_n]$:
assignment of attribute values $v_1 \in \text{dom}(A_1), \dots, v_n \in \text{dom}(A_n)$ to
attributes A_1, \dots, A_n :

- formally, can be written $v_1 \rightarrow A_1, \dots, v_n \rightarrow A_n$
- we usually write $R\langle v_1, \dots, v_n \rangle$ or even $\langle v_1, \dots, v_n \rangle$
- example tuples over $Tutorials[ID, site, tutorial, topic]$:

$\langle 1, w3schools, SQL_2003STD, Databases \rangle$

$\langle 2, w3schools, HTML_5, WebDev \rangle$

Convenient notation (for later): if t is a tuple over $R[A_1, \dots, A_n]$
and $\{A'_1, \dots, A'_m\} \subseteq \{A_1, \dots, A_n\}$ then $t[A'_1, \dots, A'_m]$ is the
projection of t to $\{A'_1, \dots, A'_m\}$:

$\langle 2, w3schools, HTML_5, WebDev \rangle[ID, topic] = \langle 2, WebDev \rangle$

When $m = 1$, we can write $t.A$ instead of $t[A]$

Formally: relations and databases

Relation state over relation schema $R[A_1, \dots, A_n]$: a finite **set** I of tuples over $R[A_1, \dots, A_n]$

- 'set' means no order of elements, no repetitions
- may be written $I \subseteq \text{dom}(A_1) \times \dots \times \text{dom}(A_n)$
(but formally not correct for us due to the order mismatch)

Relation: a relation schema + relation state

Database: finite set of relations (i.e., essentially, tables) with different names

Variants of the relational model

Many technical **non-essential variations** can be found:

- **unordered** vs. **ordered** signature (i.e., attributes, columns)
- **named** perspective (for attributes) vs. **unnamed**:
 - if the signature is ordered, names of attributes are **not important** and may be ignored (not the case for tables!)
 - moreover, attribute domains may be **abstracted** away (i.e., only one big domain is considered)
 - relation schema is just **name and arity** (written as R^2 or $R(\cdot, \cdot)$)
 - inspired by **formal logic** (which is foundations for databases):
tuples are logical **facts** $R(a, b)$

Other details are (very) **essential**:

- **no order**, **no repetitions** of tuples in a relation, etc. (see below)

Semantics (the meaning) of databases

Intuitive: **tuples** represent **certain facts** that correspond to a real-world relationships

Tutorials(1, *w3schools*, *SQL_2003STD*, *Databases*)

means that there is a **SQL_2003STD** tutorial on **Databases** topic on **w3schools** site with ID **1** (of course, the fact may not hold, but the database creators **claim** it)

Closed world assumption is assumed in the relational databases: all facts **not mentioned** in the database, are claimed **not to hold**.

No tuple *Tutorials*(1, *w3schools*, *Mars*, *Astronomy*)—no tutorial.

Why to bother? Allows to intuitively answer many queries (e.g., queries with **NOT EXISTS**)

Different in other database models: **RDF** knowledge graphs, etc.

3. Integrity constraints

Key integrity constraints: intuition

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Attribute *ID* is **special** in this relation: has **no repeated** values

- called a **(candidate) key** (of the relation)

Question: any other keys in this relation?

Key integrity constraints: intuition

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Attribute *ID* is **special** in this relation: has **no repeated** values

- called a **(candidate) key** (of the relation)

Not a **coincidence**: we *want* it to be a real ID

- can require *ID* in schema *Tutorials*[*ID*, *site*, *tutorial*, *topic*] so that each relation over this schema has *ID* as key (i.e., enforce **key constraint**)
- use **PRIMARY KEY** or **UNIQUE** declaration in SQL

Both notions generalise to **attribute sets**

Candidate and super keys formally

Super key of a relation (schema + state): a subset X of the attributes of the relation such that if t_1 and t_2 are two different tuples then $t_1[X] \neq t_2[X]$

- **observation**: the relation signature is **always** a super key

Candidate key of a relation: a **minimal** super key

- that is, removing any attribute causes the remaining attributes to no longer be a super key

Candidate and super key examples

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Super keys: $\{ID\}$, $\{ID, site\}$, $\{site, tutorial\}$,
 $\{ID, site, tutorial, topic\}$, etc.

Not super keys: $\{site\}$, $\{tutorial, topic\}$, etc.

Candidate keys: $\{ID\}$, $\{site, tutorial\}$ **Not unique!**

Not candidate keys: $\{ID, site\}$, $\{tutorial, topic\}$, etc.

Key constraints formally

Extend our relational model notions:

Relation schema with a key constraint: a relation name R with attributes A_1, A_2, \dots, A_n , and a key subset of A_1, A_2, \dots, A_n

Relation over $R[A_1, A_2, \dots, A_n]$ **satisfies** the key constraint if the key subset is a super key

A relation schema may have one or several key constraints declared; one is called **primary key** constraint (or just **primary key**)

- written as $R[\underline{A_1}, A_2, \dots, \underline{A_n}]$, where the underlined attributes are the key subset (double underline is also used)
- for example *Tutorials*[ID], *site*, *tutorial*, *topic*]

Question: Do you remember how to create a primary key in SQL?
Other (candidate) key constraint?

Primary key examples

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Relation over *Tutorials*[*ID*, *site*, *tutorial*, *topic*] (*ID* is a super key)

Not relation over *Tutorials*[*ID*, *site*, *tutorial*, *topic*] (*site* is not a super key)

Relation over *Tutorials*[*ID*, *site*, *tutorial*, *topic*]

Observation: since all attributes are always a super key, we can assume that if a primary key for a schema is not declared, then it is silently all the attributes

Foreign key (referential) integrity constraints intuition

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Students:

<i>name</i>	<i>tutorialID</i>
Alice	1
Alice	2
Bob	2

Values of attribute *tutorialID* are **not arbitrary** in relation *Students*: all values are assumed to be mentioned in *ID* of *Tutorials*

- can declare *tutorialID* to be a **foreign key** in *Students*[*name, tutorialID*] **referring** to the primary key *ID* of *Tutorials*[*ID, site, tutorial, topic*] so that the two relations in the database are required to be consistent in this sense

This notion generalises to **attribute sets**

Foreign key (referential) constraints formally

Sorry, a bit **technical**; but the idea is simple

Attributes B'_1, \dots, B'_k of relation over $S[B_1, \dots, B_m]$ are a **foreign key** referring to a candidate key A'_1, \dots, A'_k of relation over $R[A_1, \dots, A_n]$ if every tuple t_1 over $S[B_1, \dots, B_m]$ **refers** to a tuple t_2 over $R[A_1, \dots, A_n]$ —that is, $t_1[B'_1, \dots, B'_k] = t_2[A'_1, \dots, A'_k]$

Foreign key constraint is a pair of sets of attributes B'_1, \dots, B'_k and A'_1, \dots, A'_k in relation schemas as above such that A'_1, \dots, A'_k is declared as key and $dom(B'_1) = dom(A'_1), \dots, dom(B'_k) = dom(A'_k)$

Relations over $S[B_1, \dots, B_m]$, $R[A_1, \dots, A_n]$ **satisfy** this constraint if B'_1, \dots, B'_k is a foreign key referring to A'_1, \dots, A'_k for relations

A database (set of relations) can come with **foreign keys**, which should be **satisfied**

Foreign key (referential) constraints examples

Tutorials:

<u>ID</u>	site	tutorial	topic
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Students:

name	tutorialID
Alice	1
Alice	2
Bob	2

Satisfy the foreign key constraint where *tutorialID* of *Students* refers to *ID* of *Tutorials*

Foreign key (referential) constraints examples

Tutorials:

<u>ID</u>	site	tutorial	topic
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Students:

name	tutorialID
Alice	1
Alice	2
Bob	2
Bob	25

Do not satisfy the foreign key *tutorialID* of *Students* referring to *ID* of *Tutorials*

Sometimes, foreign keys are allowed to refer **only to primary keys**

- see our example

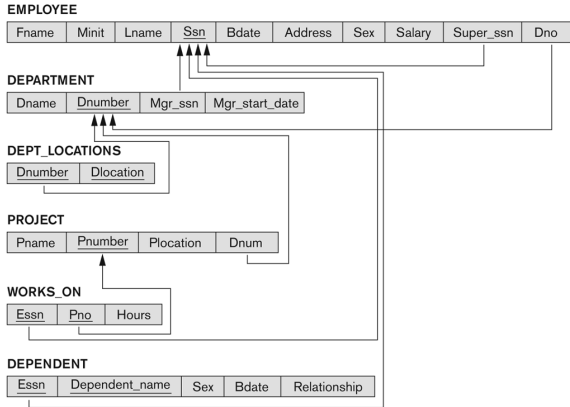
Corresponding attributes **do not need to** have the same name

- see our example

It is allowed to refer to the **same table**

- imagine a table of employees with an *ID* and *managerID* attributes (the big boss has to manage himself/herself)

Complex schema example



Copyright © 2016 Ramez Elmasri and Shamkant B. Navathe

Foreign keys are convenient to draw with **arrows**

In this figure, all foreign keys consist of **one attribute** (simple case)

Who needs to care about keys?

Question: Does a casual database user who writes a query over a database need to know about keys?

Answer: Not really

Question: Does a database manager who inputs data into a database need to know about keys?

Answer: Yes, of course

Question: Does a DBMS programmer who writes a (`SELECT`) query engine need to bother about keys?

Answer: Yes, because keys are the knowledge that may help a lot to evaluate queries efficiently, **and we will see how**

What we have learned

1. History of data models
2. Classic relational model (a version of)
3. Integrity constraints (keys and foreign keys)

Next time: SQL introduction (SQL data model, SQL sublanguages)

IN3020&4020 – Database Systems (2024)

Part: Intro & Recap

Lecture 3: SQL Summary – 1

22 January

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Materials to read

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapters 6&7
2. *Database systems – the complete book* by H. Garcia-Molina, J.D. Ullman, and J. Widom (2nd ed.): Section 6
3. PostgreSQL 14 tutorial: Part II
<https://www.postgresql.org/docs/current/sql.html>
4. IN2090 – Databaser og datamodellering (Leif Harald Karlsen):
Lectures 5, 6, 10, 11
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h23/timeplan/>
5. etc.

The **difference** between concrete SQL systems and versions are **not essential** today: we look at basics

Purpose of the lecture

I assume that you know SQL to some extent

- IN2090 (or equivalent) is a **prerequisite**

Knowledge of all details (esp., practical) are **not important** for us

- we learn the **principles of how** SQL works
- **not** the details of SQL **usage**

Today's lecture is largely a **sync-up**

- by **no means** a comprehensive SQL tutorial
- if you do not understand what I am talking about,
come back to IN2090 (not much really)
- you can **play with PostgreSQL** (group sessions for details)

But also has material **beyond** IN2090

1. Data Model of SQL
(including differences with relational model)
2. SQL Intro
3. SQL DDL and DML
4. SQL DQL (next time)

1. Data Model of SQL (including differences with relational model)

SQL is based on the **relational model**, but there are some **differences** between its data model and the relational model

Question: What are the main differences?

Answer: two main differences

1. **NULLs**
2. repeated and ordered tuples in **output** relations

In SQL, there is a special value `NULL`, that can appear in tuples as usual domain values

The meaning of `NULL` is manifold:

- unknown, not applicable, not exists, etc.

The effect on the data model is `minor`:

- just assume that `NULL` belongs to all domains of attributes
- special constraint on attributes can forbid `NULL` values
- usually, `NULL` is forbidden for primary keys
(but allowed for `UNIQUE`)
- foreign keys have a special treatment of `NULL`

SQL NULLs

In **SQL**, there is a special value **NULL**, that can appear in tuples as usual domain values

The meaning of **NULL** is manifold:

- **unknown**, **not applicable**, **not exists**, etc.

The effect on the query language is **dramatic**:

- very **non-intuitive** and causes a lot of **confusion** for all types of users (e.g., are two **NULLs** equal or not?)
- requires a lot of special non-trivial treatment in **implementation** and often slows down **performance**
- we will **not** pay much attention to **NULL** since there is **nothing** conceptually interesting for our studies of the **main principles**

In the passing on NULLs

In 2009, Sir C. A. R. (or Tony) Hoare said:

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Bag (multiset) relations example

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

SQL query `SELECT Tutorials.site FROM Tutorials` gives

π_{site} **Tutorials:**

<i>site</i>
w3schools
w3schools
w3schools
w3resource
w3resource

Input: relation. Output: **not relation.**

Question: Why?

Bag (multiset) relations motivation 1

SQL input: relations. SQL output: **not necessarily relation.**

Question: Why?

Answer 1: duplicate elimination is computationally expensive

- if a user does not care, better to keep the duplicates
- if they care, they can use DISTINCT

`SELECT DISTINCT Tutorials.site FROM Tutorials` gives

$$\delta(\pi_{site} \mathbf{Tutorials}):$$

<u>site</u>
w3schools
w3resource

Bag (multiset) relations motivation 2

SQL input: relations. SQL output: **not necessarily relation.**

Question: Why?

Answer 2: duplicates are essential for aggregate queries

```
SELECT COUNT(*) AS cnt FROM  
    SELECT Tutorials.site FROM Tutorials
```

$$\frac{\gamma_{COUNT(*) \rightarrow cnt}(\pi_{site} \mathbf{Tutorials})}{cnt}$$

$$5$$

Input & output: **relations.** Intermediate: **not relation**

Essential to define (compositional) semantics

- in this particular query, we can rewrite the an equivalent query that manipulates only relations
- this is not always the case **Homework:** Can you come up to an example?

Bag (multiset) relational model formally

What we need to change in our formalisation? Surprisingly little:

Bag relation state over relation schema $R[A_1, \dots, A_n]$: a finite **bag** I of tuples over $R[A_1, \dots, A_n]$

- ‘bag’ (a.k.a. **multiset**) means no order of elements, but repetitions are allowed (**observation**: every set is a bag)
- $\{a, b, c\}$ is a set, $\{\{a, a, b, c\}\} = \{\{a, b, a, c\}\}$ is a bag

Bag relation: a relation schema + bag relation state

Bag database: finite set of bag relations with different names

- usual relations are sometimes called **set** relations to contrast

Observation: super keys (including candidate keys) are possible only for set relations

- cannot silently assume that every relation schema has a primary key
(for sets, it is all attributes)

Relation (and bag relation)

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Not a relation, but bag relation

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
1	w3schools	SQL_2003STD	Databases
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases

Same bag relation

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
4	w3resource	SQL_2003STD	Databases
3	w3schools	CSS_3	WebDev

Ordered tables in SQL

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

SELECT * FROM Tutorials ORDERED BY topic gives

OrderedTutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev

Output is the **same** relation as input,
but it seems that we are **missing** something...

Ordered relations (and ordered bag relations) in SQL

One may argue that `ORDER BY` only influences the **output**, and can be **ignored** on the intermediate levels of query evaluation. If so, we could essentially abstract away this **'presentational'** issue. Unfortunately, this is **not true**; query

```
SELECT DISTINCT Tutorials.topic FROM
```

```
    SELECT * FROM Tutorials ORDERED BY topic LIMIT 3
```

gives only *Databases*, but

```
SELECT DISTINCT Tutorials.topic FROM
```

```
    SELECT * FROM Tutorials ORDERED BY topic DESC LIMIT 3
```

gives both *Databases*, *WebDev*

Do we need to reconsider our data model again?

Do we need to reconsider our data model again? Yes :(

But we **will not** in this course:

- such queries are **rare** and **not very meaningful**
- ordered relational model is **technical**, but **straightforward**
- this model is **not essential** for the rest of this course
- we generally concentrate on **set relational model** as the most **fundamental** for databases, mentioning **bags** when necessary

2. SQL Intro

SQL is a query language for relational databases

- (Arguably) a major reason for the **success** of relational databases
- Is a practical rendering of **relational algebra** (see next lectures)
- Usually reads as 'sequel' for historical reasons
- **Now** popularly known as 'Structured Query Language'
- Regularly updated **ANSI&ISO standards** since 1986 (newest is SQL:2016)
- Many SQL-based **DBMS systems** (Oracle, MS SQL Server, MySQL, PostgreSQL)
- The **differences** between the standards and systems are generally **inessential** for this course

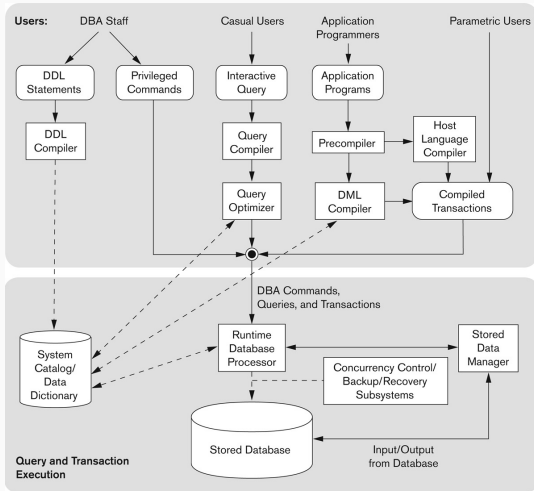
SQL components

SQL has several sub-languages:

- **Data Definition Language (DDL)**: language for relation schemas and constraints (**CREATE**, **ALTER**, **DROP**, etc.)
- **Data Manipulation Language (DML)**: language for populating relational databases (**INSERT**, **DELETE**, etc.)
- **Data Query Language (DQL)**: language for querying relational databases (**SELECT**)
- Data Control Language (DCL): language for access and user management (**GRANT**, **REVOKE**) – later in the course
- ...

DQL is sometimes considered as a part of **DML**

The place of SQL in DBMS architecture



SQL: the language of users for interaction with data

3. SQL Data Definition and Data Manipulation Languages

Data Definition Language (DDL)

```
CREATE TABLE Tutorials (  
    ID int PRIMARY KEY,  
    site text NOT NULL,  
    tutorial text,  
    topic text );
```

```
CREATE TABLE Students (  
    name text,  
    tutorialID int,  
    startdate date,  
    FOREIGN KEY tutorialID REFERENCES Tutorials(ID) );
```

Creates database schema consisting of two relation **schemas**
(with no instances)

Tutorials [ID, site, tutorial, topic]

Students [name, tutorialID, startdate]

with a foreign key (constraint) from the **second** to the **first**

Data Definition Language (DDL)

```
ALTER TABLE Students  
ADD COLUMN sID int;
```

```
ALTER TABLE Students  
ADD PRIMARY KEY (sID);
```

Changes schema to

Students [ID, name, tutorialID, startdate]

```
DROP TABLE Tutorials CASCADE;
```

Removes both tables from the database (both because of `CASCADE`)

Main datatypes

Largely **system-dependent**:

- not every type is implemented by every **DBMS**
- **allocated space** may be also different

Basic datatypes:

- **numeric**: `INT` (= `INTEGER`), `FLOAT` (= `REAL`), etc.
- **char-string**: `CHAR(n)`, `VARCHAR(n)`, `TEXT`, etc.
- **Boolean**: `BOOLEAN` (three-valued: `TRUE`, `FALSE`, `NULL`)

Complex datatypes:

- **date&time**: `DATE` (e.g., yyyy-mm-dd), `TIME`, `TIMESTAMP`, etc.
- **enums**: `ENUM(...)`
- **arrays**: `INTEGER[n]`, `TEXT[] []` (more like lists)

Not particularly important for us

Data Manipulation Language (DML)

Schema: Tutorials [ID, site, tutorial, topic]

Manual data input:

```
INSERT INTO Tutorials
VALUES (1, w3schools, SQL_2003STD, Databases),
      (2, w3schools, HTML_5, WebDev),
      ...
      (5, w3resource, MySQL, Databases);
```

May use `SERIAL`, `DEFAULT` in DDL statements to simplify input

Input using DQL (see below):

```
INSERT INTO Tutorials
SELECT ...
```

Input may **fail** due to **type mismatch**, **constraint violation** (need transactions!), etc.

Data Manipulation Language (DML)

Schema: Tutorials[ID, site, tutorial, topic]

Tuples may be updated and deleted in a similar way as retrieved using DQL (see below):

```
UPDATE Tutorials
SET tutorial = 'Mars', topic = 'Astronomy'
WHERE ...
```

```
DELETE Tutorials
WHERE ...
```

(**DELETE** removes **tuples** from tables, **not tables** themselves, cf. **DROP TABLE**)

May also **fail** due to **constraint violation** (need transactions!), etc.

3. SQL Data Query Language

3.1. Overview

Data Query Language (DQL): basics

DQL:

- the most important part of SQL (and whole DBMS)
- saying SQL we often mean the DQL sub-language
- realised via **SELECT** queries with basic **syntax**

```
SELECT <attribute list>  
FROM <table>  
[ WHERE <condition> ] ...
```

- ‘<table>’ is a **relation name** or **another query** (we do not need to write the **whole relation** here, only the name)
- ‘<attribute list>’ is a list of attributes in the schema of the relation referred by ‘<table>’

Question: What is the **semantics**?

Answer: SQL query **inputs** a database (set of relations) and **outputs** a bag(!) relation according to some rules (see examples below)

Query example

Schema (underline is a primary key here):

Tutorials[ID, site, tutorial, topic]

Students[name, tutorialID, startdate]

Query (select-project-join):

```
SELECT s.name, t.tutorial           ← projection
FROM Students s JOIN Tutorial t     ← join
    ON s.tutorialID = t.ID
WHERE t.topic = 'Databases';       ← selection
```

retrieves all student-tutorial pairs on topic databases

IN3020&4020 – Database Systems (2024)

Part: Intro & Recap

Lecture 4: SQL Summary – 2

23 January

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Materials to read

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapters 6&7
2. *Database systems – the complete book* by H. Garcia-Molina, J.D. Ullman, and J. Widom (2nd ed.): Section 6
3. PostgreSQL 14 tutorial: Part II
<https://www.postgresql.org/docs/current/sql.html>
4. IN2090 – Databaser og datamodellering (Leif Harald Karlsen): Lectures 5, 6, 10, 11
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
5. etc.

The **difference** between concrete SQL systems and versions are **not essential** today: we look at basics

Purpose of the lecture

I assume that you know SQL to some extent

- IN2090 (or equivalent) is a **prerequisite**

Knowledge of all details (esp., practical) are **not important** for us

- we learn the **principles of how** SQL works
- **not** the details of SQL **usage**

Today's lecture is largely a **sync-up**

- by **no means** a comprehensive SQL tutorial
- if you do not understand what I am talking about,
come back to IN2090 (not much really)
- you can **play with PostgreSQL** (group sessions for details)

But also has material **beyond** IN2090

1. Data Model of SQL (last time)
2. SQL Intro (last time)
3. SQL DDL and DML (last time)
4. SQL DQL
 - query syntax and semantics
 - constructs and keywords
 - basic and advanced examples
 - ...

4. SQL Data Query Language

4.1. Overview

DQL:

- the most important part of SQL (and whole DBMS)
- saying SQL we often mean the DQL sub-language
- realised via `SELECT` queries with basic **syntax**

```
SELECT <attribute list>  
FROM <table>  
[ WHERE <condition> ] ...
```

- '<table>' is a **relation name** or **another query** (but we do not need to write the **whole relation** here, only the name)
- '<attribute list>' is a list of attributes in the schema of the relation referred by '<table>'

Question: What is the **semantics**?

Answer (for SQL): **SQL** query **inputs** a database (set of relations) and **outputs** a bag(!) relation according to some rules (see below)

Query example

Schema (underline is a primary key here):

Tutorials [ID, site, tutorial, topic]

Students [name, tutorialID, startdate]

Query (select-project-join):

```
SELECT s.name, t.tutorial           ← projection
FROM Students s JOIN Tutorials t   ← join
    ON s.tutorialID = t.ID
WHERE t.topic = 'Databases';       ← selection
```

retrieves all student-tutorial pairs on topic databases

4.2 SELECT clause

SELECT clause comments

```
SELECT s.name, t.tutorial
```

...

- Selects **columns** from the (bag) relation constructed below
- Corresponds to **projection** in relational algebra (see next lectures), so `s.name`, `t.tutorial` are called **projected** attributes
- Relation name **qualification** (`s.`, etc.) may be omitted if no ambiguity
- Shortcut `*` may be used for **all** attributes
- By default, returns a **bag relation** (with possible duplicates)
- **DISTINCT** can be used for **duplicate elimination**
- **ORDER BY**, **LIMIT**, etc. can be used at the end of the query for **arranging tuples** in the output
- Can do **value invention**:

```
SELECT s.ID + 5, d AS Day(t.startdate)
```

3.3. FROM clause

FROM clause comments

...

```
FROM Students s JOIN Tutorials t ON s.tutorialID = t.ID
```

...

- constructs a **relation** from other relations
- in general, can contain an **arbitrary query** (e.g., another **SELECT** query)
- in the most simple case, **FROM** clause is a **relation name**
- (possibly nested) **operations** can be used: **JOIN**, **UNION**, **INTERSECT**, etc.
- the most common is **JOIN** operation (and its variations)

JOIN (syn. INNER JOIN): example

Combines tuples from two relations in all possible ways that agree on declared attributes

```
Tutorials [INNER] JOIN Topics ON Tutorials.topicID = Topics.ID
```

		Tutorials:				Topics:		
		<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>ID</i>	<i>topic</i>	
on	1	w3schools	SQL_2003STD	1	1	Databases	gives	
	2	w3schools	HTML_5	2	2	WebDev		
	3	w3schools	CSS_3	3				
	4	w3resource	SQL_2003STD	1				

Tutorials ⋈ **Topics:**

<i>Tutorials.ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>Topics.ID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	1	Databases
2	w3schools	HTML_5	2	2	WebDev
4	w3resource	SQL_2003STD	1	1	Databases

OUTER JOINS : example

Combines tuples from two relations in all possible ways that agree on declared attributes; if left (or right) does not match anything, filled with nulls

Tutorials LEFT [OUTER] JOIN Topics ON Tutorials.topicID=Topics.ID

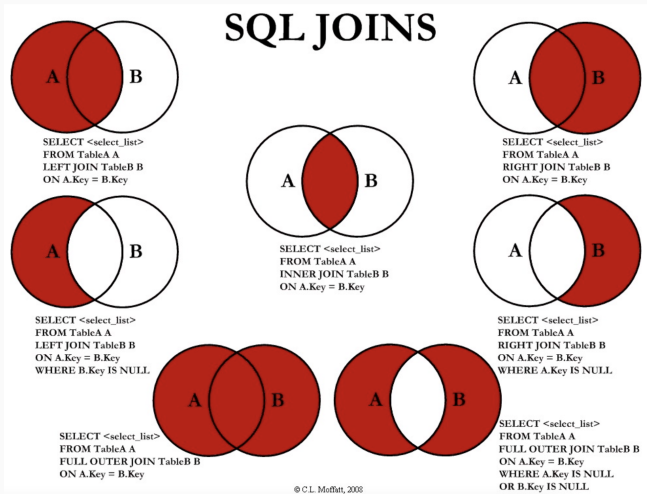
Topics RIGHT [OUTER] JOIN Tutorials ON Tutorials.topicID=Topics.ID

	Tutorials:				Topics:		
	<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>ID</i>	<i>topic</i>	
on	1	w3schools	SQL_2003STD	1	1	Databases	gives
	2	w3schools	HTML_5	2	2	WebDev	
	3	w3schools	CSS_3	3	25	Asronomy	
	4	w3resource	SQL_2003STD	1			

Tutorials ⋈ **Topics:**

<i>Tutorials.ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>Topics.ID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	1	Databases
2	w3schools	HTML_5	2	2	WebDev
3	w3schools	CSS_3	3	NULL	NULL
4	w3resource	SQL_2003STD	1	1	Databases

Joins (confusing) visualisation



Question: What does the area in the circles represent?

Self-join exercise

It is common to join a table with itself (directly or indirectly).

Schema: Employees [Id, Name] and Managers [empId, mgrId]

Query: 'Give the names of each employee and his/her manager'

A (very) naive **wrong** solution:

```
SELECT e.Name, e.Name
FROM Employees e JOIN Managers
    ON empId=Id AND mgrId=Id;
```

We need two 'copies' of Employees

Self-join exercise

It is common to join a table with itself (directly or indirectly).

Schema: Employees [Id, Name] and Managers [empId, mgrId]

Query: 'Give the names of each employee and his/her manager'

A **correct** solution:

```
SELECT e.Name, s.Name
FROM Employees e JOIN Managers
      ON empId=e.Id JOIN Employees s
      ON s.Id = mgrId;
```

Set- (bag-) theoretic operations

Relations (i.e., tables) are **sets** (or **bags**) of tuples

We can apply **usual set-theoretic** operations to them:

- set-theoretic (**duplicates eliminated**): UNION, EXCEPT, INTERSECT
- bag-theoretic (**duplicates preserved**): UNION ALL, EXCEPT ALL, INTERSECT ALL

Example:

```
SELECT * FROM Employees UNION Interns;
```

Notes:

- **complex queries** can be used as arguments
- arguments must have the **same schema**

Subqueries

We can use subqueries in the `WHERE` clause:

```
SELECT ... FROM
    ... (SELECT ... FROM Table WHERE ...) ...;
```

Subqueries can be **correlated**:

```
SELECT * FROM Tab1 JOIN
    (SELECT Tab2.a FROM Tab2 WHERE Tab1.b+5 > Tab2.c) ON Tab1.a=Tab2.a;
```

Silly query in this case, may be a lot more **difficult**

Often **rewritable** into usual joins, but **not always**

Homework: Try to come to an example

Hint: play with bags

WITH-subqueries and views

Subqueries can be used inside other queries
(we will see more examples)

We can **give them names** for convenience:

```
WITH SubqueryName1 AS (SELECT ...),  
     SubqueryName2 AS (SELECT ...)  
SELECT ... SubqueryName1 ... SubqueryName1 ... ;
```

If a subquery is used in several queries, we can give it a name **only once**—that is, create a **virtual relation**:

```
CREATE VIEW View1 AS (SELECT ...);
```

We can then use **View1** in other queries as usual tables (relations)

But **no table** is created in reality, it is still just a name of a **subquery**

Typical **recursive query** (<recursive-query> can use **Relation**):

```
WITH RECURSIVE Relation (Att1, ..., Attn) AS (  
    <non-recursive-query> UNION [ALL] <recursive-query>  
) SELECT ... FROM Relation ...;
```

Evaluation procedure:

1. **Relation** is initialised with the evaluation of <non-recursive-query>
2. <recursive-query> is evaluated on the **current state** of **Relation**, **union** is taken the **current state** of **Relation**, and the **result** is loaded into **Relation**
3. Step 2 is **repeated** until there is no change (i.e., **fix-point state** is reached)
4. Outer **SELECT** is evaluated on the **fix-point state** of **Relation**

Typical **recursive query** (<recursive-query> can use **Relation**):

```
WITH RECURSIVE Relation (Att1, ..., Attn) AS (  
    <non-recursive-query> UNION [ALL] <recursive-query>  
) SELECT ... FROM Relation ...;
```

Observations:

- corresponds to **fix-point** extension of first-order logic
- cannot be expressed via other operators
- **Relation** is getting bigger (more tuples) at each step
- **Homework question:** Does this process always stop?

Recursive SQL (classic) example

Schema: FlightRoutes [fromCity, toCity]

Query: Find all cities you can fly to from Oslo
(with any number of flight changes)

Solution:

```
WITH RECURSIVE Destinations (destCity) AS (  
    SELECT f.toCity, FROM FlightRoutes f  
        WHERE f.fromCity = 'Oslo'  
    UNION  
    SELECT rf.toCity FROM Destinations d  
        JOIN FlightRoutes rf ON d.destCity = rf.fromCity  
) SELECT * FROM Destinations;
```

Question: Why does this always terminate? **Answer:** Outputs only cities mentioned in the database

Cannot be done with **usual** join queries, because the number of stops is **not bound** in advance

3.4. WHERE clause

WHERE clause comments

...

```
WHERE t.topic = 'Databases';
```

- Selects **tuples** (rows) from the relation constructed in the FROM clause according to the condition in the clause
- Corresponds to **selection** in relational algebra
- The condition can mention both constant values (e.g., 'Databases'), which are the same for every tested tuple, and attributes (e.g., `t.topic`), which are instantiated by the attribute value for each tuple
- Values can be combined to **expressions** by functions (+, external, etc.), Boolean operators (<, **LIKE**, etc.), logical operators (**AND**, **OR**, **NOT**), case expressions (**CASE WHEN ... THEN ... ELSE ... END**), etc.
- The condition can also mention **subqueries** (see below)
- Special care should be given to **NULL** (next slide)

NULL in an value with special properties

- It is **illegal** to use NULL explicitly as part of an expression
- Usual **comparison** of NULL with **any value** evaluates to FALSE
- Both `NULL = NULL` and `NOT (NULL = NULL)` are **FALSE**
(formally, **three-valued logic** is involved)
- **Special comparison operators** can be used:
`X IS NULL`, `X IS NOT NULL`, `X IS DISTINCT FROM Y`
(work with NULL as expected)
- Own rules for **grouping** & **aggregation** with NULL (see below)

`COALESCE(Expr1, Expr2, ...)` returns the first argument that is not NULL, and NULL if all are NULL

- Typical use: `COALESCE(Address, 'not given')`

`NULLIF(Expr1, Expr2)` returns NULL if the arguments evaluate the same, or Expr1 otherwise

Conditions with subqueries: EXISTS expression

EXISTS(SELECT ...): **Boolean** (TRUE/FALSE) condition that checks existence of answers of the subquery

- evaluates to TRUE if the query gives non-empty result

Example:

```
SELECT p.name FROM Persons p
       WHERE EXISTS(SELECT ...);
```

Question: Does it look a bit silly?

Can you come to an example of ... so the query makes sense?

Answer: The subquery can be **correlated** with the outside (as usual subqueries)

```
SELECT p.name FROM Persons p
       WHERE EXISTS(SELECT * FROM Addresses a WHERE p.name = a.name);
```

Conditions with subqueries: SOME

`SOME(SELECT ...)` (syn.: `ANY(SELECT ...)`) looks for matching values in the subquery:

- example query (best **explanation**)

```
SELECT p.name FROM Persons p
      WHERE p.savings > SOME(
          SELECT pq.savings FROM Persons pq WHERE pq.age = 30));
```

- **meaning**: all people with savings more than **at least** one 30-years-old person

Can be used against multiple attributes via `ROW`:

```
... WHERE ROW(Tab.a,Tab.b) > SOME( two-column table here )
```

`IN` is a special case of `SOME`:

- ... `IN(SELECT ...)` synonym of ... = `SOME(SELECT ...)`

Conditions with subqueries: ALL

From the previous slide for comparison:

- example query:

```
SELECT p.name FROM Persons p
       WHERE p.savings > SOME(
           SELECT pq.savings FROM Persons pq WHERE pq.age = 30));
```

- **meaning**: all people with savings more than **at least** one 30-years-old person

ALL(SELECT ...) 'symmetric' to ALL:

- example query (best **explanation**):

```
SELECT p.name FROM Persons p
       WHERE p.savings > ALL(
           SELECT pq.savings FROM Persons pq WHERE pq.age = 30));
```

- **meaning**: all people with savings more than **all** 30-years-olds

Test our skills

Schema: Projects [prID, prName, customerID, prStartDate, ...]
Customers [cID, cName, cAddress, ...]
Employees [eID, eName, eTitle, eStartDate, ...]
Timesheets [eID, tDate, prID, hours, ...]

Query: Find names and titles of all employees who worked on **at least one** project started after 2014 and was ordered by customer 'Alice'

'Divide-and-Conquer' solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (  
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID  
WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Who was on which projects

```
WITH EmployeeOnProjects AS (  
SELECT e.eName,e.eTitle,t.prID FROM Employees e JOIN Timesheets t ON t.eID=e.eID)
```

- Put the two together

```
SELECT eop.eName,eop.eTitle  
FROM EmployeeOnProjects eop JOIN CorrectProjects cp ON eop.prID=cp.prID;
```

Test our skills: more challenging

Schema: Projects [prID, prName, customerID, prStartDate, ...]
Customers [cID, cName, cAddress, ...]
Employees [eID, eName, eTitle, eStartDate, ...]
Timesheets [eID, tDate, prID, hours, ...]

Query: Find names and titles of all employees who worked on **all projects** started after 2014 and was ordered by customer 'Alice'

Solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (  
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID  
WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Employees that do **not** have any (correct project they did **not** worked on):

```
SELECT e.eName,e.eTitle FROM Employees e WHERE NOT EXISTS (  
SELECT * FROM CorrectProjects cp WHERE NOT EXISTS (  
SELECT * FROM Timesheets t WHERE t.prID=cp.prID AND t.eID=e.eID));
```

Test our skills: more challenging

Query: Find names and titles of all employees who worked on **all projects** started after 2014 and was ordered by customer 'Alice'

Solution:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (  
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID  
WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Employees that do **not** have any (correct project they did **not** worked on):

```
SELECT e.eName, e.eTitle FROM Employees e WHERE NOT EXISTS (  
SELECT * FROM CorrectProjects cp WHERE NOT EXISTS (  
SELECT * FROM Timesheets t WHERE t.prID=cp.prID AND t.eID=e.eID));
```

Observations:

- Innermost **SELECT** is executed again for each employee and each project (**twice correlated query**)
- Thus, can be **slow**
- Could use **other mechanisms** (counting, LEFT JOIN, etc.)

3.5. Aggregation

Aggregation

A mechanism to accumulate values from many tuples into one using an aggregation function (`COUNT`, `COUNT(*)`, `MAX`, `MIN`, `SUM`, etc.)

Example:

```
SELECT p.Name, COUNT(m.Id) AS mCount
FROM Persons p JOIN Movies m ON p.Id=m.ActorId WHERE m.DirectorId=1234
GROUP BY p.Id, p.Name
      HAVING mCount > 10;
```

Question: What is the difference between `WHERE` and `HAVING`?

Evaluated in **steps**:

1. `FROM-WHERE` (i.e., the query 'inside', without `SELECT`)
2. the tuples divided into groups, one group for each tuple of grouping attributes (i.e., those in `GROUP BY`)
3. aggregated values are calculated for each group, giving a new table with one tuple per group
4. the whole thing is filtered through the `HAVING` condition clause
5. the attributes in `SELECT` are projected

Aggregation

A mechanism to accumulate values from many tuples into one using an aggregation function (COUNT, COUNT(*), MAX, MIN, SUM, etc.)

Example:

```
SELECT p.Name, COUNT(m.Id) AS mCount
FROM Persons p JOIN Movies m ON p.Id=m.ActorId WHERE m.DirectorId=1234
GROUP BY p.Id, p.Name
HAVING mCount > 10;
```

Question: What is the difference between WHERE and HAVING?

Comments on aggregation:

- all non-aggregates in the SELECT clause need to be in the GROUP BY clause
- aggregates can not be used in WHERE (only in HAVING)
- without GROUP BY, we always have exactly 1 row:
aggregates for the whole table

NULL-values are ignored in aggregation

- even in the case of `COUNT`
- exception is `COUNT(*)`

Aggregation functions will return `NULL` if evaluated over an empty set of values (after the `NULLs` are eliminated)

- exception is `COUNT` and `COUNT(*)`

Test our skills: more challenging with aggregates

Schema: Projects [prID, prName, customerID, prStartDate, ...]
Customers [cID, cName, cAddress, ...]
Employees [eID, eName, eTitle, eStartDate, ...]
Timesheets [eID, tDate, prID, hours, ...]

Query: Find names and titles of all employees who worked on **all projects** started after 2014 and was ordered by customer 'Alice'

Solution with aggregates:

- All projects with the correct conditions:

```
WITH CorrectProjects AS (  
SELECT p.prID FROM Projects p JOIN Customers c ON p.customerID=c.cID  
WHERE p.prStartDate>'2014-12-31' AND c.cName='Alice'),
```

- Who was on correct projects:

```
WITH UniqueEmployeesOnCorrectProjects AS (  
SELECT DISTINCT e.eID,e.eName, e.eTitle,t.prID FROM Employee e  
JOIN Timesheets t ON t.eID=e.eID JOIN CorrectProjects cp ON cp.prID=t.prID
```

- Put the two together:

```
SELECT uecp.eName, uecp.eTitle FROM UniqueEmployeesOnCorrectProjects uecp  
GROUP BY uecp.eID, uecp.eName, uecp.eTitle  
HAVING COUNT(uecp.prID) = SOME(SELECT COUNT(*) FROM CorrectProjects);
```


What have we learned?

A summary of the main database query language:

SQL

IN3020&4020 – Database Systems (2024)

Part 1: Query Processing

Lectures 5, 6: Relational Algebra

29, 30 January

Egor V. Kostylev


IFI, University of Oslo

egork@ifi.uio.no



We start new part: Query Processing

Syllabus of the course:

- (Extended) Intro and SQL recap 

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key **ingredients**:

Relational Algebra and Indexes

We study these ingredients **first** (almost in isolation)
and **then** put all things together

Materials to read about Relational Algebra

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 8 (**maybe, not the best**)
2. *Foundation of Databases* by S. Abiteboul, R. Hull & V. Vianu: Part B (**more details**)
available at <http://webdam.inria.fr/Alice/>
3. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 2.4
4. **IN2090 – Databaser og datamodellering** (Leif Harald Karlsen, Dimitru Roman): Lectures 3 (**basics**)
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
5. (**English**) **Wikipedia** has a good introduction
6. etc.

- Many good places with interactive exercises:
 - **w3resource** (<https://www.w3resource.com/>)
 - **w3schools** (<https://www.w3schools.com/sql/>)
 - Web pages of **systems** (MySQL, PostgreSQL, Oracle, etc.)
- They use SQL rather than RA
- Should **not be a problem**: for simple examples
 - syntactic translation is direct
 - semantic differences are immaterial

1. Role of RA in SQL engines
2. Algebras and algebraic laws
3. Relational Algebra intro
4. Core RA operations
5. Algebraic laws for core operations
6. (Some) expressible operations

1. Role of RA in SQL engines

SQL query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⎵ parsing

(parse tree)

⎵ preprocessing

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarsIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$$

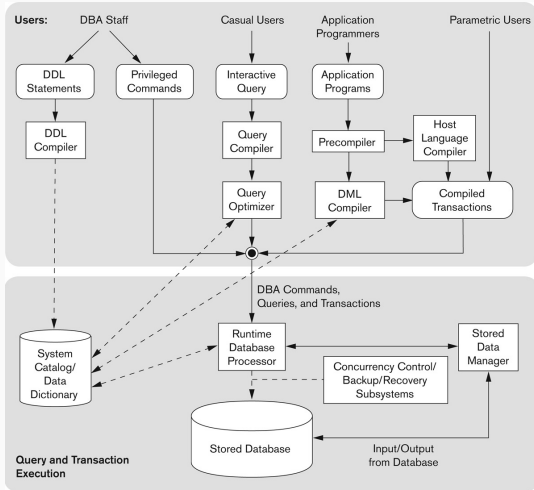
⎵ optimisation

$$\pi_{title}(\mathbf{StarsIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$$

⎵ evaluation over tables (a lot of things)

(Result)

The place of RA in DBMS architecture



2. Algebras and Algebraic Laws

Definition (Algebra)

- **Domain** D : collection of values
- **Operations**: functions from D^k to D (k is arity of the function)

- **Expressions**:
 - **Atomic**: elements of D
 - **Complex**: operations applied to other expressions
(e.g., $f(3, g(5))$)
- Expressions **evaluate** to elements of D

- Infix notation is often used for binary ($k = 2$) operations:
 - for example, instead of $+(3, 5)$ we write $3 + 5$
 - brackets or conventions used: $(3 + 5) \times 2$ vs. $3 + 5 \times 2$

Example: Integer arithmetic

- Domain: **Integers** $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$
- Operations: usual $+$, $-$, \times , $/$ (all binary)

- Expression examples:

$2 + 5$ evaluates to 7

$((2 - 4) \times 5) + (8 / 2)$ evaluates to -6

$8 / 3$ evaluates to **undefined**

Example: Regular expressions (the algebra of)

- Domain: Sets of strings over an alphabet (set of symbols) Σ
- Operations:
 - binary ($k = 2$): \circ (or nothing, concatenation), $|$ (alternation)
 - unary ($k = 1$): $*$ (Kleene star)
- Expression examples (for $\Sigma = \{a, b\}$):
 - ab^* evaluates to $\{a, ab, abb, abbb, \dots\}$
 - $(a | b)^*ab$ evaluates to all words ending with ab(Note: here, a and b are abbreviations for $\{a\}$ and $\{b\}$)
- Expressible operations:
 - ? ($e? := \epsilon | e$ for every domain element e)
 - + ($e^+ := ee^*$ for every domain element e)

Definition (Algebra, reminder)

- **Domain** D : collection of values
- **operations**: functions from D^k to D (k is arity of the function)

Algebraic Law: an identity (or equivalence) that holds in an algebra

Examples:

- $x + y = y + x$, $(x \times y) / y = x$ (for all integers x, y)
are laws for Integer Algebra
- $(e_1 e_2) e_3 = e_1 (e_2 e_3)$, $e^+ = e e^*$ (for all sets of strings)
are laws for RegExpr Algebra

Named Algebraic Laws

Common laws may have **names**:

- binary **op** is **commutative** if $e_1 \text{ op } e_2 = e_2 \text{ op } e_1$ (for all e_1, e_2)
- binary **op** is **associative** if $(e_1 \text{ op } e_2) \text{ op } e_3 = e_1 \text{ op } (e_2 \text{ op } e_3)$
- binary **op** is **idempotent** if $e \text{ op } e = e$
example: $\max(a, a) = a$ for integers
- unary **op** is **idempotent** if $\text{op}(\text{op}(e)) = \text{op}(e)$
(not a coincidence, but no need for you to know why)
example: $(e^*)^* = e^*$ for sets of words
- binary **op** is **left-distributive** over binary **op'** if
 $e_1 \text{ op } (e_2 \text{ op}' e_3) = (e_1 \text{ op } e_2) \text{ op}' (e_1 \text{ op } e_3)$
example: $a \times (b + c) = (a \times b) + (a \times c)$ for integers
when **op** is commutative, we have just **distributivity**

3. Relational Algebra Intro

Variants of Relational Algebra

- There are **many** RAs. Two RA may be
 - **Equivalent**: some operations are **expressible** via each other (e.g., \times is expressible via \bowtie and back in the presence of σ and ρ)
 - One **strictly more expressive** than another: one has operations **non-expressible** in the other
 - Have **different domains** (e.g., **named/unnamed**, **sets/bags**)
- Full SQL requires a **very expressive** variant of RA for **bags** (Turing complete!)
- We start with a **basic** RA for **sets** under **named** perspective (as suggested by Codd)
- Why (except historic importance)?
 - every set is a bag
 - all set operations may be simulated by bag operations
 - fundamental core, which also covers **most of real SQL queries**

(Our) Relational Algebra domain

Question: What is the domain of RA?

- Domain: (Set) relations (i.e., tables) up to their names
- Relation components (reminder):
 - Relation name: just a string (this we abstract away)
 - Signature: set of attribute names with domains (i.e., datatypes)
 - Set (no repetitions!) of relation tuples (a.k.a. records, rows): assignments of values to attribute names conforming signature
- Example:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

(Our) Relational Algebra domain

Question: What is the domain of RA?

- Domain: (Set) relations (i.e., tables) up to their names
- Relation components (reminder):
 - Relation name: just a string (this we abstract away)
 - Signature: set of attribute names with domains (i.e., datatypes)
 - Set (no repetitions!) of relation tuples (a.k.a. records, rows): assignments of values to attribute names conforming signature
- Same example:

R:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

(Our) Relational Algebra operations

- Domain: Relations (up to their names)
- Core operations:
 - (set) **union** \cup
 - (set) **difference** \setminus
 - **selection** σ_C
 - **projection** π_{Atts}
 - **Cartesian product** \times (a.k.a. **cross product**, **cross join**)
 - **renaming** ρ_N (technical, often not mentioned)
- Some unary, some binary, usually not always applicable
- Some **parametrised**: families of operations rather than single
- Expressible operations:
 - (set) **intersection** \cap
 - other **joins**: **natural** \bowtie , **theta-** \bowtie_θ , **semi-** \bowtie & \bowtie , etc.
 - **division** \div
 - ...

4. Core RA operations

Union: binary, no parameters (example)

Set-theoretic union of tuples in same-structured relations:

Tutorials1:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev

Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

U

Tutorials1 \cup Tutorials2:

=

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Union: binary, no parameters (definition)

Set-theoretic union of tuples in same-structured relations

Union (formal definition)

- Let R and S be two relations
- **Union** $R \cup S$ is the relation that
 - defined when R and S have the same signature (i.e., same attribute names and their datatypes)
 - has this signature
 - contains each tuple that is either in R or in S (or both)

Set-theoretic union: **repetitions are removed!**

Difference: binary, no parameters (example)

Set-theoretic difference of tuples in same-structured relations:

Tutorials1:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev

Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

=

Tutorials1 \ Tutorials2:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev

Difference: binary, no parameters (definition)

Set-theoretic difference of tuples in same-structured relations

Difference (formal definition)

- Let R and S be two relations
- **Difference** $R \setminus S$ is the relation that
 - defined when R and S have the same signature (i.e., attribute names together with their datatypes)
 - has the same signature
 - contains each tuple that is in R but not in S

No repetitions anyway (by construction)

Selection: unary, one parameter (example)

Selects tuples from a relation satisfying condition in the parameter:

$$\sigma_{ID>2 \ \& \ topic='Databases'}$$

Tutorials:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

$$=$$

$\sigma_{ID>2 \ \& \ topic='Databases'}$ (Tutorials):			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Selection: unary, one parameter (definition)

Selects tuples from a relation satisfying condition in the parameter

Selection (formal definition)

- Let R be a relation
- Let C be a condition—that is, Boolean (i.e., using $\&$, \vee , \neg) combination of atoms
 - $e_1 = e_2$, $e_1 \neq e_2$, where e_1 and e_2 are attribute names of R or some **values** (numbers, strings, etc.)
 - domain specific operations over attribute names and values, e.g., $e_1 > e_2$, e *LIKE* *RegExp* (no subqueries (!!), etc.)
- **Selection** $\sigma_C(R)$ is the relation that
 - is defined when C mentions only attribute names of R and all operations in C are well-typed
 - has the same signature as R
 - contains all tuples in R satisfying C

Projection: unary, one parameter (example)

Selects columns from a relation, as specified in the parameter

$$\pi_{site, topic} \left(\begin{array}{c} \textbf{Tutorials:} \\ \hline \begin{array}{cccc} ID & site & tutorial & topic \\ \hline 1 & w3schools & SQL_2003STD & Databases \\ 2 & w3schools & HTML_5 & WebDev \\ 3 & w3schools & CSS_3 & WebDev \\ 4 & w3resource & SQL_2003STD & Databases \\ 5 & w3resource & MySQL & Databases \end{array} \end{array} \right)$$
$$= \begin{array}{cc} \pi_{site, topic}(\textbf{Tutorials}): \\ \hline site & topic \\ \hline w3schools & Databases \\ w3schools & WebDev \\ w3resource & Databases \end{array}$$

Projection: unary, one parameter (definition)

Selects columns from a relation, as specified in the parameter

Projection (formal definition)

- Let R be a relation
- Let $Atts$ be a list of attribute names
- **Projection** $\pi_{Atts}(R)$ is the relation that
 - is defined when $Atts$ mentions only attribute names of R
 - has signature $Atts$ (with domains from R)
 - contains projections (i.e., restrictions) of all tuples in R to $Atts$

Repetitions are removed!

Cartesian product: binary, no parameters (example)

Combines tuples from two relations in all possible ways:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>
1	w3schools	SQL_2003STD
2	w3schools	HTML_5
3	w3schools	CSS_3
4	w3resource	SQL_2003STD
5	w3resource	MySQL

×

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev

Tutorials × Topics:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	Databases
2	w3schools	HTML_5	1	Databases
3	w3schools	CSS_3	1	Databases
4	w3resource	SQL_2003STD	1	Databases
5	w3resource	MySQL	1	Databases
1	w3schools	SQL_2003STD	2	WebDev
2	w3schools	HTML_5	2	WebDev
3	w3schools	CSS_3	2	WebDev
4	w3resource	SQL_2003STD	2	WebDev
5	w3resource	MySQL	2	WebDev

Cartesian product: binary, no parameters (definition)

Combines tuples from two relations in all possible ways

Cartesian (or Cross) Product (formal definition)

- Let R and S be two relations
- **Cartesian product** $R \times S$ is the relation that
 - is defined when R and S have no(!) common attribute name
 - has the union of signatures of R and S as signature
 - contains all possible combinations of tuples in a tuple in R with a tuple in S
- Note: first operation that has **more** in output than in input
- Maybe really large
- Makes little sense, not used in practice much
- Useful to present **natural join** \bowtie (see below)

Renaming: unary, one parameter

Renames attributes in a relation as specified in the parameter:

$$\rho_{topic \rightarrow area} \left(\begin{array}{c} \text{ProjTutorials:} \\ \hline \begin{array}{cc} \textit{site} & \textit{topic} \\ \hline \text{w3schools} & \text{Databases} \\ \text{w3schools} & \text{WebDev} \\ \text{w3resource} & \text{Databases} \end{array} \end{array} \right) = \begin{array}{c} \rho_{topic \rightarrow area}(\text{ProjTutorials}): \\ \hline \begin{array}{cc} \textit{site} & \textit{area} \\ \hline \text{w3schools} & \text{Databases} \\ \text{w3schools} & \text{WebDev} \\ \text{w3resource} & \text{Databases} \end{array} \end{array}$$

Renaming (formal definition)

- Let R be a relation
- Let N be a list $A_1 \rightarrow A'_1, \dots, A_n \rightarrow A'_n$, where $A_1, \dots, A_n, A'_1, \dots, A'_n$ are all different attribute names
- **Renaming** $\rho_N(R)$ is the relation that
 - is defined when A_1, \dots, A_n are attribute names of R and A'_1, \dots, A'_n are not
 - is the same as R except that A_1, \dots, A_n renamed to A'_1, \dots, A'_n

Independent set of operations

Core Set RA

*Selection σ_C , Projection π_L , Cartesian Product \times ,
Union \cup , Difference \setminus , Renaming ρ_N*

- Can express **everything** in Codd's algebra (but not all SQL!)
- These are **independent** operations: dropping any of them makes the language weaker
- There are other useful **expressible** operations: intersection \cap , natural join \bowtie , division \div , etc.
 - there are **effective** algorithms for them
 - it is often **simpler** to formulate queries using them
- We discuss them after algebraic laws for core operations

5. Algebraic laws for core RA operations

RA Algebraic Laws: initial observations

The power of RA is largely in its rich set of **algebraic laws**

- allows to convert complex queries to **equivalent but simpler** ones (before evaluating them)
- 'simple' here means not 'short' or 'neat-looking', but 'expected to be evaluated quicker'
- we will discuss how to decide what is simpler (in this sense) than other **later on**
- there are (too) **many** laws, and we consider only **most prominent examples**
- start now with **core operations**, later touch expressible ones

Laws for set-theoretic operations

Union is **commutative**, **associative**, and **idempotent**:

$$R \cup S = S \cup R \quad (R \cup S) \cup T = R \cup (S \cup T) \quad R \cup R = R$$

(for all relations R, S, T)

Difference is **right-distributive** over union:

$$(R \cup S) \setminus T = (R \setminus T) \cup (S \setminus T)$$

Question: Is it **left-distributive**: $R \setminus (S \cup T) \stackrel{?}{=} (R \setminus S) \cup (R \setminus T)$?

Answer: No (come to counter-example by yourself)

Other (unnamed) laws:

$$R \setminus (S \cup T) = (R \setminus S) \setminus T = (R \setminus T) \setminus S$$

...

Laws for Cartesian product

Cartesian product is **commutative** and **associative**:

$$R \times S = S \times R \quad (R \times S) \times T = R \times (S \times T)$$

(for all relations R, S, T)

Question: is it idempotent ($R \times R \stackrel{?}{=} R$)?

Answer: Of course **no**: it is never defined

Cartesian product is **distributive** over both union and difference:

$$(R \cup S) \times T = (R \times T) \cup (S \times T)$$
$$(R \setminus S) \times T = (R \times T) \setminus (S \times T)$$

Other (unnamed) laws:

...

Laws for selection

Selection is **idempotent**:

$$\sigma_C(\sigma_C(R)) = \sigma_C(R) \quad (\text{for every relation } R \text{ and condition } C)$$

Conditions can be **reshuffled** and **split**:

$$\sigma_{C_1 \& C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$$

$$\sigma_{C_1 \vee C_2}(R) = \sigma_{C_1}(R) \cup \sigma_{C_2}(R)$$

Selection can be **pushed** through union, difference, and Cartesian product (**not always!**):

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$

$$\sigma_C(R \setminus S) = \sigma_C(R) \setminus \sigma_C(S) = \sigma_C(R) \setminus S$$

$$\sigma_C(R \times S) = \sigma_C(R) \times S \quad \text{if } C \text{ mentions only attributes of } R$$

... (other laws)

Projection is **idempotent**:

$$\pi_{Atts}(\pi_{Atts}(R)) = \pi_{Atts}(R) \text{ (for relation } R \text{ and attributes } Atts)$$

Idempotence can be generalised [corrected!]:

$$\pi_{Atts}(R) = \pi_{Atts}(\pi_{Atts, Atts'}(R))$$

Projection can be **pushed** through union and Cartesian product:

$$\pi_{Atts}(R \cup S) = \pi_{Atts}(R) \cup \pi_{Atts}(S)$$

$$\pi_{Atts_R, Atts_S}(R \times S) = \pi_{Atts_R}(R) \times \pi_{Atts_S}(S)$$

if $Atts_R$ are attributes of R and $Atts_S$ are attributes of S

... (other laws)

...

...

Homework: find a couple by yourself

6. (Some) expressible operations

Intersection: binary, no parameters

Set-theoretic intersection of tuples in same-structured relations:

Tutorials1:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev

Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

\cap

$=$

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
3	w3schools	CSS_3	WebDev

Intersection (formal definition)

Intersection: $R \cap S := R \setminus (R \setminus S)$ for every R and S
(including the undefined cases)

Natural Join: binary, no parameters (example)

Combines tuples from two relations in all possible ways that agree on common attributes:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
3	w3schools	CSS_3	2
4	w3resource	SQL_2003STD	1
5	w3resource	MySQL	1

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev
2	WebDev1

⋈

Tutorials ⋈ Topics:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	Databases
2	w3schools	HTML_5	2	WebDev
2	w3schools	HTML_5	2	WebDev1
3	w3schools	CSS_3	2	WebDev
3	w3schools	CSS_3	2	WebDev1
4	w3resource	SQL_2003STD	1	Databases
5	w3resource	MySQL	1	Databases

=

Natural Join: binary, no parameters (definition)

Natural Join (formal definition)

Let R and S be two relations:

- A_1, \dots, A_n and A'_1, \dots, A'_m their attributes
- $\{B_1, \dots, B_k\} = \{A_1, \dots, A_n\} \cap \{A'_1, \dots, A'_m\}$: common

Natural Join: $R \bowtie S := \rho_N(\pi_L(\sigma_C(\rho_{N_R}(R) \times \rho_{N_S}(S))))$ where

- N_R and N_S rename attributes of R and S apart:

$$N_R = \{A_1 \rightarrow R.A_1, \dots, A_n \rightarrow R.A_n\}$$

$$N_S = \{A'_1 \rightarrow S.A'_1, \dots, A'_m \rightarrow S.A'_m\}$$

- C is the condition matching common attributes:

$$C = (R.B_1 = S.B_1 \ \& \ \dots \ \& \ R.B_k = S.B_k)$$

- L are all attributes of R and S except the repetitions:

$$L = R.A_1, \dots, R.A_n, S.B'_1, \dots, S.B'_\ell$$

$$\text{where } \{B'_1, \dots, B'_\ell\} = \{A'_1, \dots, A'_m\} \setminus \{A_1, \dots, A_n\}$$

- N renames attributes back:

$$N = \{R.A_1 \rightarrow A_1, \dots, R.A_n \rightarrow A_n, S.B'_1 \rightarrow B'_1, \dots, S.B'_\ell \rightarrow B'_\ell\}$$

Natural Join (observations)

- Product \times is **expressible** in the presence of natural join \bowtie
 - **Question:** Any ideas how?
 - **Answer:** $R \times S := R \bowtie S$ for every relations R, S when $R \times S$ is defined
(the difference is that \bowtie is defined in more cases than \times)
- **Another core set RA:** selection σ_C , projection π_L , natural join \bowtie , union \cup , difference \setminus , renaming ρ_N
- Natural join is one of the **most used** operations in practice
- Implementing \bowtie via \times is **extremely non-effective**
- Native algorithms for \bowtie are **common** (we will see)

Theta Join: binary, one parameter (example)

Normal join + selection:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
3	w3schools	CSS_3	2
4	w3resource	SQL_2003STD	1
5	w3resource	MySQL	1

$\bowtie_{ID > topicID}$

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev

Tutorials $\bowtie_{ID > topicID}$ **Topics:**

	<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
=	3	w3schools	CSS_3	2	WebDev
	4	w3resource	SQL_2003STD	1	Databases
	5	w3resource	MySQL	1	Databases

(This is not very meaningful query,
but the operation is very useful in practice)

Theta Join: binary, one parameter (definition)

Theta Join (formal definition)

Theta Join: $R \bowtie_{\theta} S := \sigma_{\theta}(R \bowtie S)$ for relations R and S , and condition $\theta = RAtt_1 \text{ op}_1 SAtt_1 \ \& \ \dots \ \& \ RAtt_k \text{ op}_k SAtt_k$ where $RAtt_i$ and $SAtt_i$ are attributes of R and S , and $\text{op}_i \in \{=, \neq, >, <, \geq, \leq\}$ (including the undefined cases)

- **Question:** Why to bother? Why only these conditions?
- **Answer:**
 - one the one hand, very common in practice
 - one the other, we can design specialised efficient algorithms
- **Equi-Join:** theta join with $=$ as all op_i (e.g., $R \bowtie_{RAtt=SAtt} S$)
 - essentially, naturally join with incorporated renaming
 - even more common and own algorithms

Semi-joins, anti-joins: binary, no parameters

Semi-Joins (formal definition)

Left Semi-Join: $R \ltimes S := \pi_{R\text{Atts}}(R \bowtie S)$ for relations R and S where $R\text{Atts}$ are attributes of R

Right Semi-Join: $R \bowtie S := \pi_{S\text{Atts}}(R \bowtie S)$ for relations R and S where $S\text{Atts}$ are attributes of S (symmetric: $R \bowtie S = S \bowtie R$)

- **Left Semi-Join:** all tuples in R that **join** with tuples in S

Anti-Joins (formal definition)

Left Anti-Join: $R \triangleright S := R \setminus (R \bowtie S)$ for relations R and S

Right Anti-Join: $R \triangleleft S := \dots$ (symmetric: $R \triangleleft S = S \triangleright R$)

- **Left Anti-Join:** all tuples in R that **do not join** with tuples in S

Question: Where are Outer Joins? **Answer:** Next time: not in classical relational algebra, need NULLs

Division: binary, no parameters (example 1)

'Inverse' of Cartesian Product:

TutTimesTop:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	Databases
2	w3schools	HTML_5	1	Databases
3	w3schools	CSS_3	1	Databases
4	w3resource	SQL_2003STD	1	Databases
5	w3resource	MySQL	1	Databases
1	w3schools	SQL_2003STD	2	WebDev
2	w3schools	HTML_5	2	WebDev
3	w3schools	CSS_3	2	WebDev
4	w3resource	SQL_2003STD	2	WebDev
5	w3resource	MySQL	2	WebDev

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev

TutTimesTop ÷ Topics:

<i>ID</i>	<i>site</i>	<i>tutorial</i>
1	w3schools	SQL_2003STD
2	w3schools	HTML_5
3	w3schools	CSS_3
4	w3resource	SQL_2003STD
5	w3resource	MySQL

Division: binary, no parameters (example 2)

'Inverse' of Cartesian Product:

TutTimesTop:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	Databases
3	w3schools	CSS_3	1	Databases
4	w3resource	SQL_2003STD	1	Databases
5	w3resource	MySQL	1	Databases
1	w3schools	SQL_2003STD	2	WebDev
2	w3schools	HTML_5	2	WebDev
4	w3resource	PostgreSQL	2	WebDev
5	w3resource	MySQL	2	WebDev
5	w3resource	PostgreSQL	2	WevDev

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev

÷

TutTimesTop ÷ Topics:

<i>ID</i>	<i>site</i>	<i>tutorial</i>
1	w3schools	SQL_2003STD
5	w3resource	MySQL

Division: binary, no parameters (definition)

Division (formal definition)

Division: $R \div S := \pi_{Atts}(R) \setminus \pi_{Atts}((\pi_{Atts}(R) \times S) \setminus R)$ for relations R and S , where $Atts$ is the attributes of R not mentioned in S (including the undefined cases)

- Defined only when each attribute of S is an attribute of R (follows from the definition)
- Equivalently, can be defined as:
 $R \div S$ is all tuples t from $\pi_{Atts}(R)$ such that $\{t\} \times S \subseteq R$
- As expected, $(R \times S) \div S = R$ for every R and S
- However, $(R \div S) \times S$ may be different from R
- (cf. **Euclidian division**, a.k.a. **division with remainder** for integers)

(Some) algebraic laws for expressible operations

Our definitions of the expressible operations are essentially laws:

$$\text{e.g., } R \div S = \pi_{\text{Atts}}(R) \setminus \pi_{\text{Atts}}((\pi_{\text{Atts}}(R) \times S) \setminus R)$$

Intersection \cap is **commutative**, **associative**, and **idempotent**, as well as **distribute over union** \cup

Natural join is **commutative**, **associative**, and **idempotent**:

$$R \bowtie S = S \bowtie R \quad (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \quad R \bowtie R = R$$

(the most important laws in practice)

Selection can be **pushed** through join (**not always!**):

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \quad \text{if } C \text{ mentions only attributes of } R$$

(also very important)

... (other laws)

What have we learned?

Classic Relational Algebra:

core and expressible operations, algebraic laws

IN3020&4020 – Database Systems (2024)

Part 1: Query Processing

Lectures 7, 8: Relational Algebra for SQL

5, 6 February

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Materials to read about Relational Algebra (same)

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 8 (maybe, not the best)
2. *Foundation of Databases* by S. Abiteboul, R. Hull & V. Vianu: Part B (more details)
available at <http://webdam.inria.fr/Alice/>
3. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 2.4
4. IN2090 – Databaser og datamodellering (Leif Harald Karlsen, Dimitru Roman): Lectures 3 (basics)
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
5. (English) Wikipedia has a good introduction
6. etc.

Materials to Play With (same)

- Many good places with interactive exercises:
 - **w3resource** (<https://www.w3resource.com/>)
 - **w3schools** (<https://www.w3schools.com/sql/>)
 - Web pages of **systems** (MySQL, PostgreSQL, Oracle, etc.)
- They use SQL rather than RA
- Should **not be a problem**: for simple examples
 - syntactic translation is direct
 - semantic differences are immaterial

1. What is missing in the classic set RA to cover SQL?
2. More functionality for sets
3. Bag Relational Algebra intro
4. Extending and re-defining known operations
5. Bag-specific operations
6. Algebraic laws and integrity constraints

1. What is missing in
the classic set RA to cover SQL?

Variants of Relational Algebra

- There are **many** RAs. Two RA may be
 - **Equivalent**: some operations are **expressible** via each other (e.g., \times is expressible via \bowtie and back in the presence of σ and ρ)
 - One **strictly more expressive** than another: one has operations **non-expressible** in the other
 - Have **different domains** (e.g., **named/unnamed**, **sets/bags**)
- We considered **basic RA for sets** under **named** perspective (as suggested by Codd)
- Full SQL requires a **very expressive** variant of RA for **ordered bags** (Turing complete!)
- We will look at a version of RA for **(unordered) bags**:
 - enough to capture **most of SQL**
 - but **not all**

Set Relational Algebra we know

- Domain: (Set) Relations
- Core operations:
 - (set) union \cup
 - (set) difference \setminus
 - selection σ_C
 - projection π_{Atts}
 - Cartesian product \times (a.k.a. cross product, cross join)
 - renaming ρ_N (technical, often not mentioned)
- Some unary, some binary, usually not always applicable
- Some parametrised: families of operations rather than single
- Expressible operations:
 - (set) intersection \cap
 - other joins: natural \bowtie , theta- \bowtie_θ , semi- $\bowtie \&$ \bowtie , etc.
 - division \div
 - ...

What is missing in our core RA to cover **all** SQL functionality?

1. **Value invention** (`SELECT R.A+5 FROM R`)
2. **NULLs** (in particular, for `OUTER JOINS`):
3. Recursion (`WITH RECURSIVE`) not covered
4. **Bag** relations and **aggregates** (`SELECT SUM(R.A) FROM R`)
5. Ordered relations (`ORDER BY, LIMIT`) not covered

... ..

2. More functionality for sets

Generalised Projection: unary, one parameter (example)

Projection + renaming + value invention using expressions

$$\pi_{site, topic \rightarrow area, ID+5 \rightarrow IDplus}$$

Tutorials:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

$$=$$

$\pi_{site, topic \rightarrow area, ID+5 \rightarrow IDplus}$ (Tutorials):		
<i>IDplus</i>	<i>site</i>	<i>area</i>
6	w3schools	Databases
7	w3schools	WebDev
8	w3schools	WebDev
9	w3resource	Databases
10	w3resource	Databases

Generalised Projection: unary, one parameter (definition)

Selects and modifies attributes, as specified in the parameter

Generalised Projection (formal definition)

- Let R be a relation
- Let N be a list of attributes A and maps $Exp \rightarrow A'$, where Exp is an expression over attributes and values, and A, A' are attributes
- **Generalised Projection** $\pi_N(R)$ is the relation that
 - is defined when all A' are different and not attributes of R , all other mentioned attributes (incl., A) are in the schema of R , and all types are matched
 - with schema consisting of all A and A'
 - contains all tuples constructed from tuples in R as specified by N
- Repetitions are **removed**
- Just **renaming** when Exp is an attribute
- Exp may be a **constant** (not mentioning attributes), including **NULL**
- Can create **bigger** table than input

Left Outer Join: binary, no parameters (example)

Combines tuples from two relations in all possible ways that agree on common attributes, **and** extend left **dangling** tuples with **NULLs**

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
3	w3schools	CSS_3	3
4	w3resource	SQL_2003STD	1



Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
2	WebDev
25	Asronomy

Tutorials ⋈ Topics:

	<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
=	1	w3schools	SQL_2003STD	1	Databases
	2	w3schools	HTML_5	2	WebDev
	3	w3schools	CSS_3	3	NULL
	4	w3resource	SQL_2003STD	1	Databases

Outer Joins: binary, no parameters (definition)

Natural Join + dangling tuples extended with NULLs

Outer Joins (native formal definition)

- Let R and S be two relations
- **Left Outer Join** $R \bowtie S$ is the relation that
 - all common attributes of R and S have the **same** domains
 - has the **union** of schemas of R and S as **schema**
 - contains all possible **combinations** of a tuple in R with a tuple in S that agree on common attributes **and** all **dangling tuples** in R **extended** with NULLs
 - a **dangling tuple** in R is a tuple that **does not agree** with any tuple in R on common attributes
- **Right Outer Join** $R \bowtie S := S \bowtie R$
- **Full Outer Join** $R \bowtie S := (R \bowtie S) \cup (R \bowtie S)$

Expressible via the core RA + Generalised Projection
(see the **first mandatory**)

Algebraic Laws

Generalised Projection and outer joins have their **laws**. For example:

Generalised Projection can be **pushed through** union and Cartesian product:

$$\pi_N(R \cup S) = \pi_N(R) \cup \pi_N(S)$$

$$\pi_{N_R, N_S}(R \times S) = \pi_{N_R}(R) \times \pi_{N_S}(S)$$

if N_R mentions only attributes of R
and N_S mentions only attributes of S

Left outer join is **idempotent**: $R \bowtie R = R$ ($= R \bowtie R$)

But not **commutative**: $R \bowtie S \neq S \bowtie R$ (unless $R = S$)

And not **associative**: $(R \bowtie S) \bowtie T \neq R \bowtie (S \bowtie T)$

There are many more

3. Bag Relational Algebra intro

Bag Relational Algebra domain

- Domain: **Bag relations** (i.e., **bag tables**)
- Bag relation components (reminder):
 - **Relation name**: just a string
 - **Schema**: set of **attribute names** with domains (i.e., datatypes)
 - Bag (possible repetitions!) of **tuples**:
assignments of values to attribute names conforming schema
- Example:

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
1	w3schools	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Bag Relational Algebra operations

- **Domain:** Bag Relations
- Every Set Relation is a Bag Relation
- All **set** RA operations **generalise** to bags:
 - some are just **extensions** to bags that are not sets—that is, if input relations are sets then the result is **as before**:
difference \setminus , intersection \cap , selection σ_C , Cart. product \times ,
division \div , renaming ρ_N , joins ($\bowtie, \bowtie_\theta, \ltimes, \ltimes, \triangleleft, \triangleright, \bowtie\bowtie, \bowtie\bowtie, \bowtie\bowtie$)
 - others need **re-definition** even for sets (marked by sup-script b):
union \cup^b , (generalised) projection π_N^b
(we also keep the set versions \cup and π_N ,
which are **undefined** on bags that are not sets)
- Several new operations:
 - **duplicate elimination** δ , **grouping+aggregation** γ_L, \dots

4. Extending and re-defining the known operations

Bag Union: binary, no parameters (example only)

Bag-theoretic union of tuples (multiplicities are added):

Tutorials1:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
1	w3schools	SQL_2003STD	Databases
3	w3schools	CSS_3	WebDev

Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
3	w3schools	CSS_3	WebDev
3	w3schools	CSS_3	WebDev
5	w3resource	MySQL	Databases

\cup^b

Tutorials1 \cup^b Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
1	w3schools	SQL_2003STD	Databases
3	w3schools	CSS_3	WebDev
3	w3schools	CSS_3	WebDev
3	w3schools	CSS_3	WebDev
5	w3resource	MySQL	Databases

=

Difference for bags: binary, no parameters (example only)

Bag-theoretic difference of tuples in same-structured relations
(multiplicities are **subtracted**; if negative, overwritten by 0)

Tutorials1:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev

Tutorials2:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev

=

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases

Selection for bags: unary, one parameter (example only)

Selects tuples as in the set case:

$$\sigma_{ID>2 \ \& \ topic='Databases'}$$

Tutorials:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

$$=$$

$\sigma_{ID>2 \ \& \ topic='Databases'}$ (Tutorials):			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
4	w3resource	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	Databases

Generalised Projection for bags (including renaming): unary, one parameter (example only)

Also as in set case (outputs the **same** number of tuples as in input):

$$\pi_{site, topic \rightarrow area, ID+5 \rightarrow IDplus}^b$$

Tutorials:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
4	w3resource	SQL_2003STD	Databases

$$=$$
$$\pi_{site, topic \rightarrow area, ID+5 \rightarrow IDplus}^b(\mathbf{Tutorials}):$$

<i>IDplus</i>	<i>site</i>	<i>area</i>
6	w3schools	Databases
7	w3schools	WebDev
8	w3schools	WebDev
9	w3resource	Databases
9	w3resource	Databases

Natural Join for bags (including Cartesian product): binary, no parameters (example only)

Combines tuples from two relations in all possible ways that **agree** on common attributes (multiplicities are **multiplied**):

Tutorials:

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>
1	w3schools	SQL_2003STD	1
1	w3schools	SQL_2003STD	1
2	w3schools	HTML_5	2
2	w3schools	HTML_5	2

⋈

Topics:

<i>topicID</i>	<i>topic</i>
1	Databases
1	Databases
2	WebDev
25	Astronomy

Tutorials ⋈ **Topics:**

=

<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topicID</i>	<i>topic</i>
1	w3schools	SQL_2003STD	1	Databases
1	w3schools	SQL_2003STD	1	Databases
1	w3schools	SQL_2003STD	1	Databases
1	w3schools	SQL_2003STD	1	Databases
2	w3schools	HTML_5	2	WebDev
2	w3schools	HTML_5	2	WebDev

Expressible operations for bags

Many operations are expressed exactly as in the set RA:

Theta Join: $R \bowtie_{\theta} S := \sigma_{\theta}(R \bowtie S)$

Intersection: $R \cap S := R \setminus (R \setminus S)$

Others require **more elaborate** expressions to take care of **multiplicity** (not used so much for bags, so do not give them here)

Left Semi-Join: $R \ltimes S := \dots$

Right Semi-Join: $R \ltimes S := \dots$

Left Anti-Join: $R \not\bowtie S := \dots$

Right Anti-Join: $R \not\bowtie S := \dots$

Left Outer Join: $R \bowtie\!\!\!\bowtie S := \dots$

Right Outer Join: $R \bowtie\!\!\!\bowtie S := \dots$

Full Outer Join: $R \bowtie\!\!\!\bowtie S := \dots$

Division: $R \div S := \dots$

5. Bag-specific operations

Duplicate elimination: unary, no parameters (example)

Removes **duplicates**:

$$\delta \left(\begin{array}{c} \textbf{Tutorials:} \\ \hline \begin{array}{cccc} ID & site & tutorial & topic \\ 1 & w3schools & SQL_2003STD & Databases \\ 2 & w3schools & HTML_5 & WebDev \\ 3 & w3schools & CSS_3 & WebDev \\ 4 & w3resource & SQL_2003STD & Databases \\ 4 & w3resource & SQL_2003STD & Databases \\ 5 & w3resource & MySQL & Databases \end{array} \end{array} \right)$$
$$= \begin{array}{c} \delta(\textbf{Tutorials}): \\ \hline \begin{array}{cccc} ID & site & tutorial & topic \\ 1 & w3schools & SQL_2003STD & Databases \\ 2 & w3schools & HTML_5 & WebDev \\ 3 & w3schools & CSS_3 & WebDev \\ 4 & w3resource & SQL_2003STD & Databases \\ 5 & w3resource & MySQL & Databases \end{array} \end{array}$$

Duplicate elimination: unary, no parameters (definition)

Removes **duplicates**

Duplicate Elimination (formal definition)

- Let R be a relation
- **Duplicate elimination** $\delta(R)$ is the relation that
 - is always defined
 - has the same schema as R
 - contains all tuples in R but without duplicates (i.e., with all multiplicities 1)

Set operations can be **simulated** by the bag versions and δ

For example, $R \cup S = \delta(R \cup^b S)$ for every set relations R and S

More laws **below**

Grouping+aggregation: unary, one parameter (example)

Groups and aggregates according to the parameter

$\gamma_{\text{site, SUM}(ID) \rightarrow sID, \text{COUNT}(topic) \rightarrow ctopic}$

Tutorials:			
<i>ID</i>	<i>site</i>	<i>tutorial</i>	<i>topic</i>
1	w3schools	SQL_2003STD	Databases
2	w3schools	HTML_5	WebDev
3	w3schools	CSS_3	WebDev
4	w3resource	SQL_2003STD	Databases
5	w3resource	MySQL	NULL

$= \gamma_{\text{site, SUM}(ID) \rightarrow sID, \text{COUNT}(topic) \rightarrow ctopic}(\text{Tutorials}):$

<i>site</i>	<i>sID</i>	<i>ctopic</i>
w3schools	6	3
w3resource	9	1

Question: Why 3 and 1 in the last column?

Answer: Duplicates count, *NULL* does not

Aggregate functions

An aggregate function takes a **bag of values** and returns a **value**

May be also **undefined** (e.g., due to type mismatch)

There are several aggregate functions **relevant to SQL**

We consider some **examples**:

- $\text{COUNT}(\Omega)$ counts the number of elements in Ω (with multiplicities) different from NULL :

$$\text{COUNT}(\{\{a, b, b, b, \text{NULL}\}\}) = 4$$

- $\text{COUNT_DISTINCT}(\Omega)$ counts the number of distinct elements in Ω (without multiplicities) different from NULL :

$$\text{COUNT_DISTINCT}(\{\{a, b, b, b, \text{NULL}\}\}) = 2$$

Aggregate functions

An aggregate function takes a **bag of values** and returns a **value**

May be also **undefined** (e.g., due to type mismatch)

There are several aggregate functions **relevant to SQL**

We consider some **examples**:

- $\text{MIN}(\Omega)$ computes the minimum of non-*NULL* elements in Ω :
 $\text{MIN}(\{\{1, 2, 3, 3, \text{NULL}\}\}) = 1$
- $\text{MAX}(\Omega)$ computes the maximum of non-*NULL* elements in Ω :
 $\text{MAX}(\{\{1, 2, 3, 3, \text{NULL}\}\}) = 3$
- $\text{SUM}(\Omega)$ computes the sum of non-*NULL* elements in Ω :
 $\text{SUM}(\{\{1, 2, 3, 3, \text{NULL}\}\}) = 9$
- $\text{AVG}(\Omega)$ computes the average of non-*NULL* elements in Ω :
 $\text{AVG}(\{\{1, 2, 3, 3, \text{NULL}\}\}) = 2.25$

All **undefined** if Ω has anything different from numbers and *NULL*,
evaluate to *NULL* if Ω has **only** *NULL*s (or empty)

Grouping+aggregation: unary, one parameter (definition)

Groups and aggregates the values of attributes, as specified in the parameter

Grouping and Aggregation (formal definition)

- Let R be a relation and N a list of attributes A_g and maps $Agg(A) \rightarrow A'$, where Agg is an aggregate function, and A_g, A, A' are attributes
- **Grouping with Aggregation** $\gamma_N(R)$ is the relation that
 - is defined when all A' are different and not attributes of R , all other mentioned attributes (incl., A_g and A) are in signature of R , and all types (incl., in aggregates) are matched
 - with signature consisting of all A_g and A'
 - contains all tuples constructed from R as follows:
 1. tuples in R are **grouped** by distinct tuples of values of A_g
 2. for **each group**, aggregate $Agg(\Omega)$ is computed for every $Agg(A) \rightarrow A'$, where Ω is the bag of values of A in the group
 3. one tuple is constructed for **each group** from all A_g and all A' as computed values of the aggregates $Agg(A) \rightarrow A'$

Becomes **projection + duplicate elimination** when no aggregates mentioned

6. Algebraic Laws and Integrity Constraints

Algebraic Laws for bags

Almost all the rules mentioned before for sets **extend** to bags

For example, natural join is **commutative** and **associative**:

$$R \bowtie S = S \bowtie R \quad (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Selection and projection can be **pushed** through join (**not always!**):

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \quad \text{if } C \text{ mentions only attributes of } R$$

However, there are **exceptions** (e.g., $R \bowtie R \neq R$ for bags)

Bag union \cup^b and generalised projection π_N^b have **their own** laws, often **different** from their set versions \cup and π_N (e.g., $R \cup^b R \neq R$)

Dupl. elimination δ , and grouping+aggregation γ_N also have laws:

- δ can be **pushed** through join, selection, etc.:

$$\delta(R \bowtie S) = \delta(R) \bowtie \delta(S) \quad \delta(\sigma_C(R)) = \sigma_C(\delta(R))$$

- γ_N has **very few** useful laws

(i.e., allow for only **very limited** optimisation)

Integrity Constraints in RA

We can express **integrity constraints** (primary and foreign keys, etc.) using RA

Primary key: attributes A_1, \dots, A_k of relation R

Relation R **satisfies** this primary key if for every tuple t_1 in R there is no other tuple t_2 in R such that $t_1[A_1, \dots, A_k] = t_2[A_1, \dots, A_k]$

Assuming that R is a set relation, can be written as

$\sigma_C(R \bowtie \rho_N(R)) = \emptyset$, where

- N is $B_1 \rightarrow B'_1, \dots, B_m \rightarrow B'_m$ for B_1, \dots, B_m are all the attributes of R different from A_1, \dots, A_k
- C is $B_1 \neq B'_1 \vee \dots \vee B_m \neq B'_m$

For arbitrary bag relations also possible, but requires using aggregates (**Homework:** try it!)

Integrity Constraints in RA

We can express **integrity constraints** (primary and foreign keys, etc.) using RA

Foreign key: attributes B_1, \dots, B_k of relation S and attributes A_1, \dots, A_k of relation R with matching domains

Relations S and R **satisfy** this foreign key if every tuple t_1 in S has a tuple t_2 in R such that $t_1[B_1, \dots, B_k] = t_2[A_1, \dots, A_k]$

Can be written as $\delta(\pi_{B_1, \dots, B_k}(S)) \subseteq \delta(\pi_{A_1, \dots, A_k}(R))$

What have we learned?

What is missing in the classic set RA to cover (most of) SQL

IN3020&4020 – Database Systems (2024)

Part 1: Query Processing

Lectures 9, 10: Indexes

12, 13 February

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key **ingredients**:

Relational Algebra (✓) and Indexes

We study these ingredients **first** (almost in isolation)
and **then** put all things together

Materials to read about Indexes

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 17 (and some of 16)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 14 (+ 8.3 & 8.4)
3. IN2090 – Databaser og datamodellering (Leif Harald Karlsen, Dimitru Roman): Lectures 13 (basics)
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
4. (English) Wikipedia
5. etc.

You can see the effect of indexes in PostgreSQL

```
CREATE TABLE t1(id int);

INSERT INTO t1
  SELECT n*random()
  FROM generate_series(1, 10000000) AS x(n);

CREATE TABLE t2 AS (SELECT * FROM t1);

CREATE INDEX t1_ind ON t1(id);

\timing on

SELECT * FROM t1 WHERE id = 100; --> Time: 0.792 ms

SELECT * FROM t2 WHERE id = 100; --> Time: 303.022 ms
```

1. How to store relations?
2. Types of indexes
3. Multi-level indexes (how to implement indexes)
4. Indexes in practice
5. Multi-dimensional indexes (indexes on several attributes)

1. How to store relations?

Main goal of a DBMS: answer queries quickly

Ingredients for efficiency:

1. Find a good **query plan** (i.e., **RA expression**)
2. Store data (i.e., relations) to **maximise** the speed of **retrieval**:

`SELECT * FROM R WHERE R.A = 1345` (exact search)

`SELECT * FROM R WHERE R.A > 1345` (interval search)

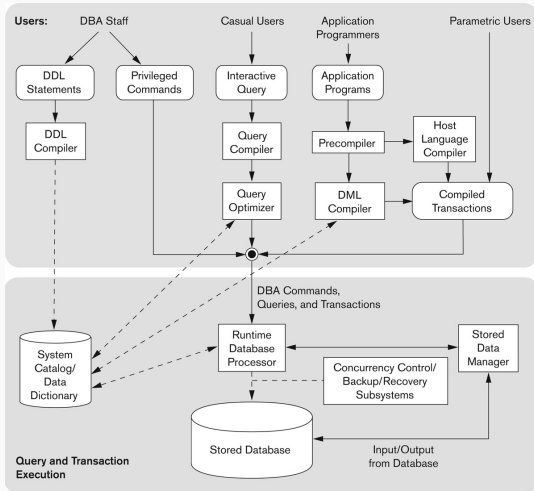
`SELECT * FROM R WHERE R.A > 1345 AND R.B = 'a'`
(multi-attribute search)

Challenge: There may be a **lot** of data (i.e., tuples)

Updates (incl. insertions and deletions) are less critical, but should also be taken into account

Today: look at how DBMSs manage with **Ingredient 2**

Place of data storage (incl. indexes) in DBMS architecture



What memory is available

Storage (memory) hierarchy:

- Primary storage:
CPU main memory, cache memory (Static RAM, DRAM)
- Secondary storage:
Magnetic disks, flash memory, solid-state drives
- Tertiary storage:
Removable media, distributed over the Web, etc.

Data in standard DBMSs is stored in secondary memory:

- good size ($\sim 10\text{--}100$ TBs now)
- slow access ($\sim 10\text{--}100$ ms now)

Secondary memory (e.g., magnetic disk):

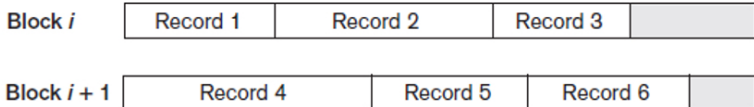
- Organised in **blocks** (~10 KBs)
- Can be **accessed** using **pointers**,
which may refer to **any point** (word) in any block
- However, **read & write** is possible only for the **whole block**
- Blocks are **consecutive**, e.g., we can retrieve
"the block in the middle between two given blocks"
(referring to blocks by their first pointers)

Storing a relation

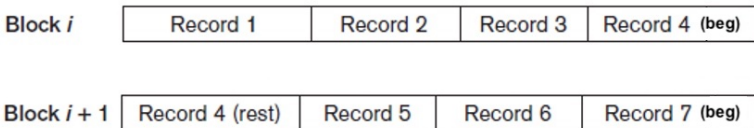
Usually, a block can accommodate **several** records (i.e., tuples)

There are options to store tuples of a **relation in a file**:

Continuous, unspanned:



Continuous, spanned:

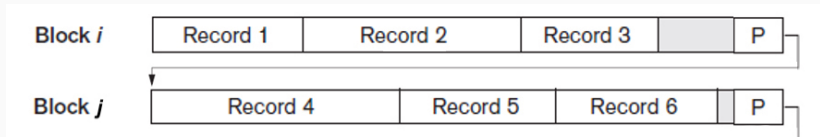


Storing a relation

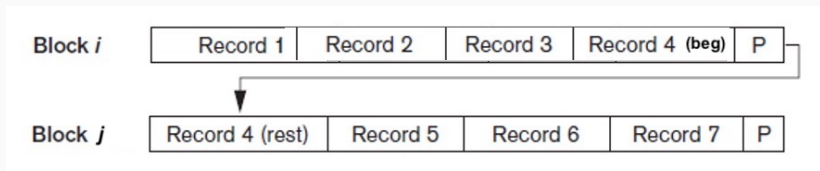
Usually, a block can accommodate **several** records (i.e., tuples)

There are options to store tuples of a **relation in a file**:

Linked, unspanned:



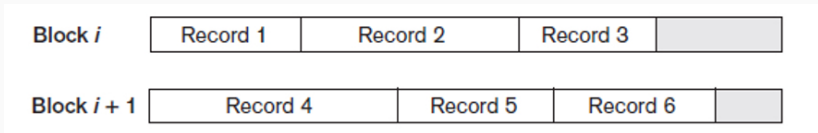
Linked, spanned:



Our assumptions

We assume:

- Generally, **continuous, unspanned**:



- (Sometimes, **linked** as well)
- all records (i.e., tuples) take the **same size**

All these assumptions are for **simplicity**,
(almost) **everything** applies to other settings

Naive tuple arrangement: (unordered) heap

Heap (or pile) file:

- tuples placed in file in order of insertion
- inserting a new tuple: very efficient
- deletion of a tuple:
 - real deletion and squeezing all following blocks (slow)
 - real deletion and squeezing only current block (efficient)
 - use deletion marker (efficient)
- searching with a given attribute value (... WHERE R.A = 1245)
 - b reads (where b is the number of used blocks)
 - $\sim b/2$ reads on average, if A is a candidate key (e.g., primary key)

Slow. What can we do?

File with tuples ordered by the **search key R.A**:

- **inserting** a new tuple: may be slow, but may use **overflow blocks** (see below)
- **deletion** of a tuple: similar to heap
- **searching** with a given key (where b is the number of used blocks)
 - $\sim b/2$ on average **naively** (even for non-keys)
 - $\leq \lceil \log_2 b + 1 \rceil$ using **binary search**

Better, but **still...** And what to do with **several attributes**?

Binary search

Let us need to find a **tuple** with key value k in an ordered relation

- try the **middle block**; if tuple with k found, **success**
- if not, identify the **half** with k **above or below** the block by checking whether the smallest key value in the block is **smaller or greater** than k
- try the **middle block** of the half; if tuple with k found, **success**
- ...

For example,

- for $b = 7$, a worst case may be retrieving blocks **#4, #2, #3** (e.g., $\lceil \log_2 b + 1 \rceil = 3$ is materialised)
- for $b = 8$, a worst case may be **#4, #6, #7, #8** (e.g., $\lceil \log_2 b + 1 \rceil = 4$ is materialised)

2. Types of Indexes

An **index** on an attribute A (in a relation R):
a data structure that **facilitates** (i.e., speeds up) finding the tuples
in R with a certain value of A

Attribute A is called **the search key**

It is possible to have an index on **every attribute** (and **every set of**),
but there is a **trade-off**:

More indexes for the same relation means

- **faster** search
- **slower** updates & **more** storage

Usually, DBMSs **automatically** create an index on every primary key

Indexes on other attributes can be created **manually**

Types of Indexes

We will consider:

Primary Index (dense or sparse)

- the tuples are **sorted** (physically) on the search key
- at most **one tuple** for a search-key value (e.g., primary key)

Cluster Index (dense or sparse)

- the tuples are **sorted** (physically) on the search key
- **any number of tuples** for a search-key value

Secondary Index (dense only)

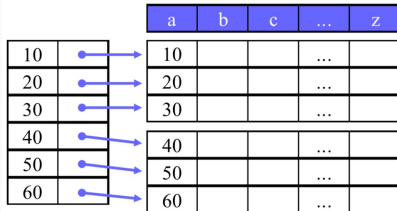
- the tuples are not **sorted** on the search key
- may be used **in addition** to other indexes on the same relation

Primary Dense Index

Primary Dense Index on the primary (or candidate) key attribute A in a relation R stored as sorted by A :

- **sorted table** with two “attributes”:
 - primary **key value**, **pointer** to the tuple with this value
- **each** tuple in R has a pair in the index

Example:



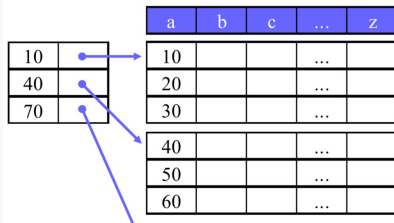
Benefit: index itself is also stored in secondary memory, but with much fewer blocks

Primary Sparse Index

Primary Sparse Index on the primary key attribute A in a relation R stored as sorted by A :

- **sorted table** with two “attributes”:
 - primary **key value**, **pointer** to the tuple with this value
- **only the first** tuple in each block of R has a pair in the index

Example:



Benefit: index itself is also stored in secondary memory, but with even fewer blocks

Retrieval comparison

Assume that we have:

- 1000000 tuples of 300B each, 4B per search key (primary), 4B per pointer
- 4K block size, 1ms for fetching a block:
 - 13 tuples per block (i.e., 76924 unspanned blocks)
 - 512 indexes per block (i.e., 1954 blocks for a dense index and 151 blocks for a sparse index)

Using no index, no order:

$$76924/2 = 38462 \text{ block accesses on average } (\sim 38.5\text{s!})$$

Using no index, but order (and binary search; think why +1 here and below):

$$\lceil \log_2(76924 + 1) \rceil = 17 \text{ block accesses at most}$$

Using primary dense index:

$$\lceil \log_2(1954 + 1) \rceil + 1 = 11 + 1 = 12 \text{ block accesses at most}$$

Using primary sparse index:

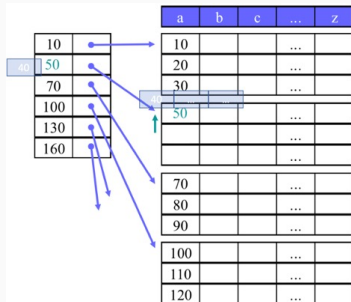
$$\lceil \log_2(151 + 1) \rceil + 1 = 8 + 1 = 9 \text{ block accesses at most}$$

Record deletion with primary sparse index

Deletion (and insertion) of tuples needs care

Delete entry with $a = 60$:
index doesn't need an update

Delete entry with $a = 40$:
index needs to be updated
(see to the right)



One usually prefers to **compress** the data in the **blocks**

One can also compress the **whole file**,
but we may like to keep free slots for **future inserts**

Dense is similar

Record insertion with (sparse) primary index

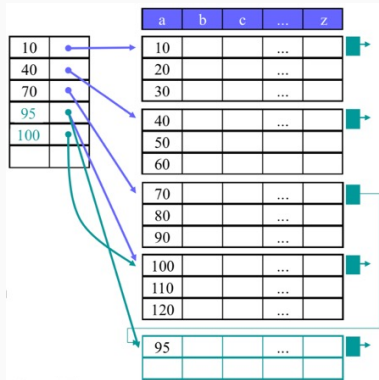
Insertion (same as deletion) of tuples needs care

If there is **space**, similar

Insert entry where $a = 95$:

No **space**. Insert a **new block**
overflow: change / no change
for index
normal: change to index,
breaks continuity

Dense is similar

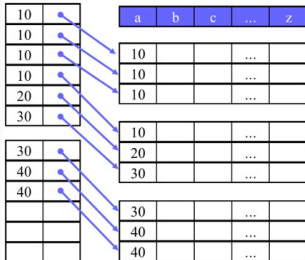


Clustering Dense Index

Clustering Dense Index on an attribute A (any!) in a relation R stored as sorted by A :

- **sorted table** with two “attributes”:
 - attribute value, pointer to the tuple with this value
- **each** tuple in R has a pair in the index

Example:



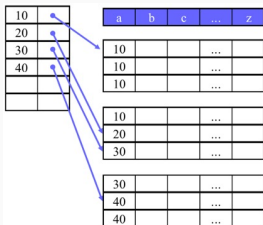
Not much different from primary dense index

Clustering Sparse Index (variant 1)

Clustering sparse index (var. 1) on an attribute A (any!) in a relation R stored as sorted by A :

- sorted table with two “attributes”:
 - attribute value, pointer to the tuple with this value
- only one index for each value of A in R has a pair in the index

Example:



Less index and faster search (good for small number of search values)

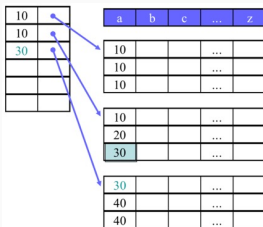
But finding all tuples for a value is more complex

Clustering Sparse Index (variant 2)

Clustering sparse index (var. 2) on an attribute A (any!) in a relation R stored as sorted by A :

- sorted table with two “attributes”:
 - attribute value, pointer to the tuple with this value
- only the first tuple in each block of R has a pair in the index

Example:



Even less index and faster search (if the number of blocks is less than the number of values)

Finding all tuples for a value is easier

Secondary Index

Secondary index on an attribute A in a relation R (not sorted by A):

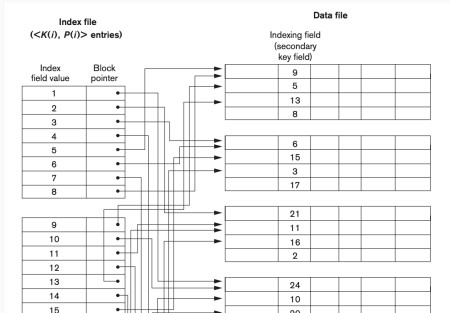
- **sorted table** with two “attributes”:
attribute value, pointer to the tuple with this value
- **every** tuple in each block of R has a pair in the index

Provide **secondary** means of accessing tuples

Used when some **other** primary access exists

Duplicates are allowed (not in the figure, may be optimised by a **level of indirection**)

Always **dense**



Dense vs Sparse indexes

	DENSE	SPARSE
Space required	One index field per tuple	One index field per data block
Block access	"Many"	"Few"
Access to entry	Direct access	Must search in the data block
Usage	All cases	Not on unsorted elements
Updates	Always updated if entry sequence is changed	Updated only if the first entry of the data block is changed

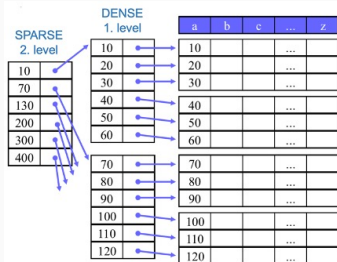
3. Multi-level Indexes

(Static) Multilevel Index

(Static) Multilevel Index on an attribute A in a relation R

- index **on** an index
- useful when the main index occupy **several blocks**

Example:



Second layer makes sense **only if** sparse

In principle, **any number of layers** is possible

Retrieval comparison

Assume that we have:

- 1000000 tuples of 300B each, 4B per search key, 4B per pointer
- 4K block size, 1ms for fetching a block:
 - 13 tuples per block (i.e., 76924 unspanned blocks)
 - 512 indexes per block (i.e., 1954 blocks for a dense index and 151 blocks for a sparse index)

No index, no order: $76924/2 = 38462$ block accesses

No index, but with order (and binary search): $\lceil \log_2 76925 \rceil = 17$ block accesses

Primary dense index: $\lceil \log_2 1955 \rceil + 1 = 11 + 1 = 12$ block accesses

Using primary sparse index: $\lceil \log_2 152 \rceil + 1 = 8 + 1 = 9$ block accesses

Using two-level index (dense 1st level, sparse 2nd level):

need $1954/512 = 4$ blocks for the 2nd level

$\lceil \log_2(4 + 1) \rceil + 1 + 1 = 5$ block accesses at most

(Static) Multilevel Indexes

Looks good, but

Problems:

1. Insertion and deletion are messy and difficult
2. How to decide how many levels we need?

Solution: Dynamic Multilevel Indexes

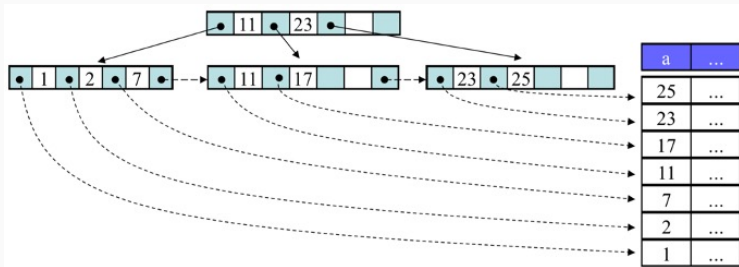
based on B-trees, B+-trees (B*-trees, etc.)

“B” stays for “balanced”

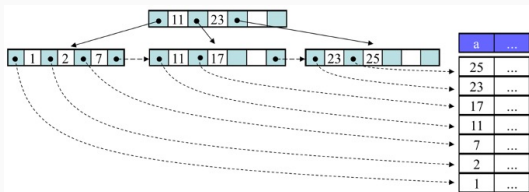
Dynamic Multilevel Indexes: B+-trees

B+-tree of order n :

- **Balanced**: all leaf nodes are at the same level
- each node has at least $\lceil (n + 1)/2 \rceil$ and at most n search keys and $+1$ pointers (except the root)
- **Inner node**: all pointers are to sub-trees, key values specify values in sub-trees as in static case
- **Leaf node**: data-pointers (dense or sparse) and 1 next-pointer



Dynamic Multilevel Indexes: B+-trees



Search with B+-trees:

- from the root to leaves—that is, the number of block accesses is the height (usually, low; e.g., for $n = 340$ and 3 levels we can refer to $>16M$ tuples)
- interval search is also fast

Insertion & deletion with B+-trees:

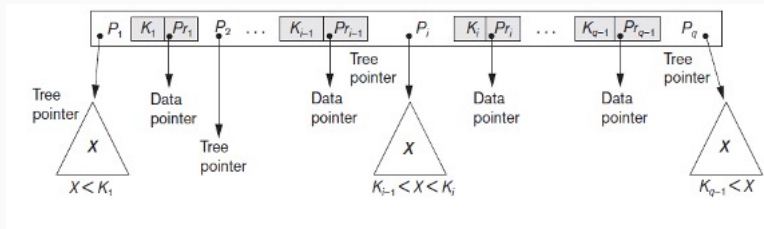
- need to ensure $\lceil (n + 1)/2 \rceil$ and n bounds
- there are special smart algorithms (Check them!)
- the height and number of nodes may change

Dynamic Multilevel Indexes: B-trees

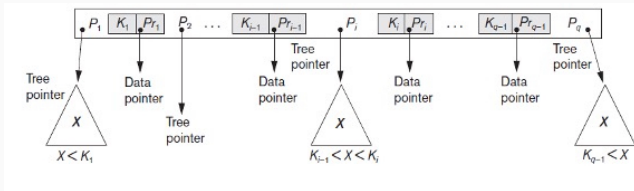
B-tree of order n

(same as B+, except that the inner nodes have pointers to data):

- all leaf nodes are at the same level
- each node has at least $\lceil (n + 1)/2 \rceil$ and at most n search keys with pointers and $+1$ sub-tree pointers (except the root, which may have less, and leaves, where “sub-tree” pointers are nulls)
- **Inner node**: interleaving pointers are to sub-trees and data
- **Leaf node**: only keys and data-pointers (+ sub-tree nulls)



Dynamic Multilevel Indexes: B-trees



Search with B-trees:

- from the root to leaves—that is, the number of block accesses is **at most** the height (better than for B+, but the height might be larger)
- interval search is also fast

Insertion & deletion with B-trees:

- need to ensure $\lceil (n+1)/2 \rceil$ and n bounds
- there are smart algorithms (**harder** and **slower** than for B+; **check them!**)
- the **height** and **number of nodes** may **change**

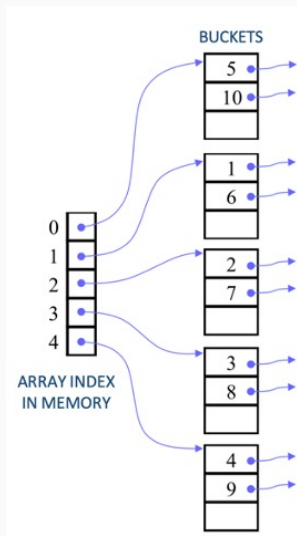
Another approach: Hash-based index

Hash-based index with hash function h (essentially, two-level index with one second-level block, with hash-values instead of search keys):

- first-level blocks are called **buckets**
- keys in buckets are **not consecutive**
- second-level block stores all possible hash values $h(\text{key})$
- bounded by the **size of a block**

Common hash function $\text{mod}(\text{key}, \text{base})$:

- **base** is usually a prime number
- ensures good **distribution** over buckets



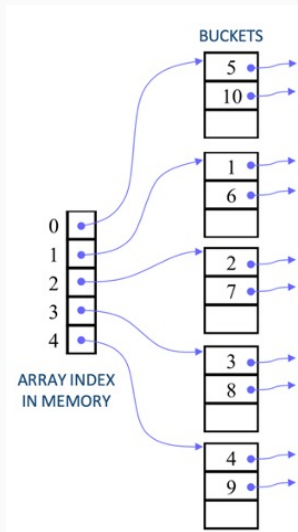
Hash-based index: Pros and cons

Hash-based index pros:

- fewer block accesses than with B & B+ for exact search (e.g., R.A = 1245)
(hash values are shorter than keys)
- updates are easy when there is space

Hash-based index cons:

- poor for interval search (e.g., R.A > 1245)
- what if there is no space in the bucket on update?
 - overflow buckets
 - dynamic hashes (extensible, linear)
(check it out!)



4. Indexes in practice

Indexes in DBMSs

Usually, DBMSs **automatically** create an index on every primary key
Indexes on other attributes can be created **manually**

The syntax is **system dependent**; for example:

```
CREATE INDEX NAME ON relName ( attName )
```

```
CREATE INDEX NAME ON relName ( attName1, . . . , attNameN )
```

(see the meaning **below**)

```
DROP INDEX indexName
```

Only some (e.g., professional) DBMSs allow us to specify the **details** of created index—that is, the **type** (e.g., B-tree, hashing), its **parameters** (e.g., B-tree order, hash function), etc.

Default index type in PostgreSQL is **B-tree**

<https://www.postgresql.org/docs/14/indexes.html>:

“B-trees can handle equality and range queries on data that can be sorted into some ordering”

PostgreSQL also has **GiST**:

a mechanism for implementing indexes for complex data types.

You can see the effect of indexes in PostgreSQL

```
CREATE TABLE t1(id int);

INSERT INTO t1
  SELECT n*random()
  FROM generate_series(1, 10000000) AS x(n);

CREATE TABLE t2 AS (SELECT * FROM t1);

CREATE INDEX t1_ind ON t1(id);

\timing on

SELECT * FROM t1 WHERE id = 100; --> Time: 0.792 ms

SELECT * FROM t2 WHERE id = 100; --> Time: 303.022 ms
```

5. Multi-dimensional indexes

What to do with multi-attribute search?

```
SELECT * FROM R WHERE A = 30 AND B < 5
```

Naive strategy:

1. Use an index on A to fetch all tuples with $A = 30$
2. Filter the tuples $B < 5$

Not very efficient: a lot of unnecessary block accesses

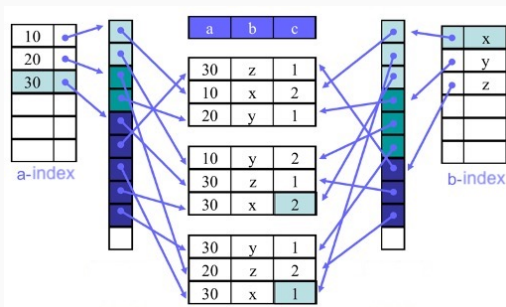
Question: Any ideas how to do it better?

Multi-attribute search: Better strategy

`SELECT * FROM R WHERE A = 30 AND B < 5`

More advanced strategy:

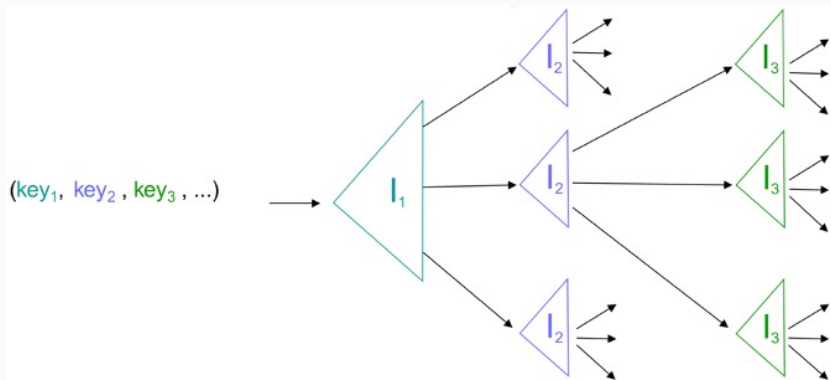
1. find all pointers(!) to tuples with $A = 30$ using a dense index for A
2. find all pointers(!) to tuples with $B < 5$ using a dense index for B
3. intersect the sets of pointers and fetch the relevant tuples



But again: two dense indexes does not look like the best solution

Multiple-Key Indexes

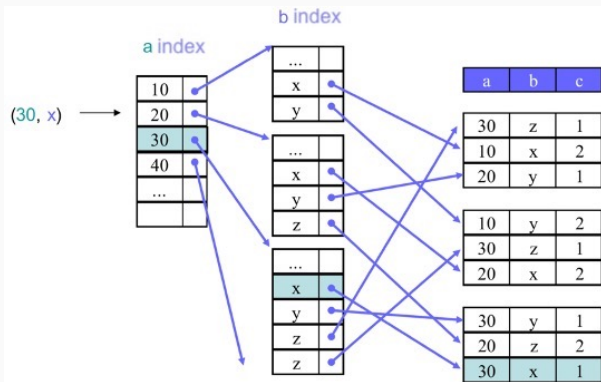
Multiple-Key index (hierarchical multi-dimensional approach):
an own index for inferior dimensions for every exterior keys



Multiple-Key Index: Example

`SELECT * FROM R WHERE A = 30 AND B = 'x'`

Search first for $A = 30$, then for $B = 'x'$ in the index for $A = 30$



Multiple-Key indexes: When to use

This index is **effective** for:

SELECT * FROM R WHERE A = 30 AND B = 'x' ✓

SELECT * FROM R WHERE A = 30 AND B > 'x' ✓

SELECT * FROM R WHERE A = 10 ✓

SELECT * FROM R WHERE B = 'x' ✗

SELECT * FROM R WHERE A >= 30 AND B = 'x' ✗

Would work with other dimension order (**another index**)

But there are many other **alternatives**:

- Other multidimensional **tree-like** structures
- Multidimensional **hash-like** structures
- **Bitmap** indexes

Multidimensional search visualisation

We can visualise two-dimensional search as a map:

```
SELECT * FROM R WHERE a = 30 AND b = 'x'
```

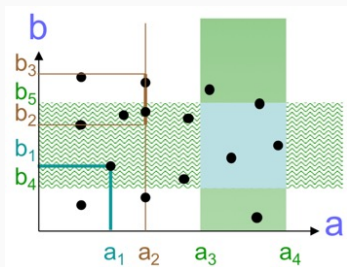
point

```
SELECT * FROM R WHERE a = 30 AND b >= 'x' AND b <= 'y'
```

segment

```
SELECT * FROM R WHERE a >= 30 AND a <= 40 AND b >= 'x' AND b <= 'y'
```

area



(Advanced) Multidimensional tree structures

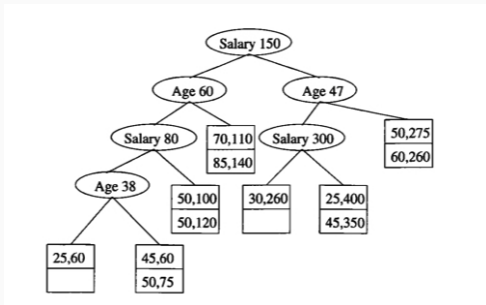
There are **tree-like structures** for multidimensional search, **inheriting** from B(+)-trees in some ways (**balancing**, **easy updates**, etc.):

- **k-d trees**: a binary search tree in which intermediate nodes are splitting values for individual dimensions every leaf node is a block of k-dimensional points
- **Quad-trees**: a tree data structure where each internal node has exactly four children (used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions)
- **R-trees**: tree data structures used for spatial access methods (i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons)
- ...

k-d trees: idea

k-d trees: a **binary search tree** in which intermediate nodes are splitting values for individual dimensions every leaf node is a block of *k*-dimensional points (dimensions 'circulate' along a branch)

Illustration:

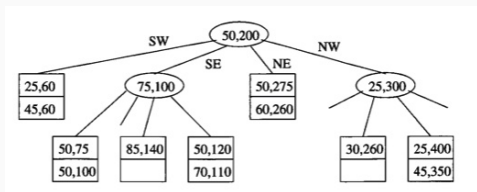
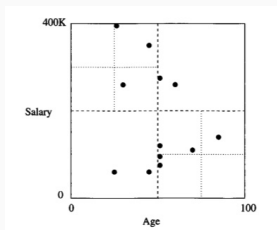


Useful for range and **nearest neighbour** searches

Quad-trees: idea

Quad-tree: a tree data structure where each internal node has exactly four children (often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions)

Illustration:



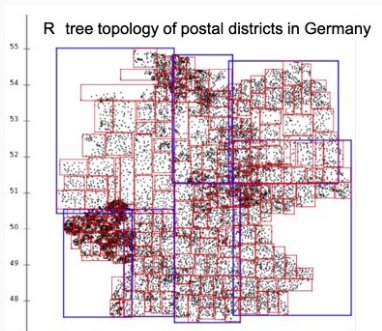
R-trees: idea

Balanced tree, where nodes group close points

- inner node: smallest rectangle containing all subtree points (may overlap)
- leaf node: point (with a pointer to a tuple)

Search (incl. intersection, contained in, nearest neighbour): easy

Insertion: challenging



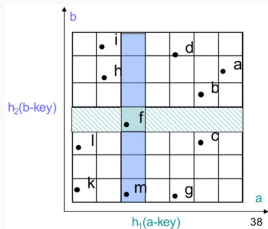
Multidimensional hash-like structure: Grid files

Grid files:

- 'Hash' function h_d for each dimension d individually
- 'Hashes' to the position number in the d -dimension of the **matrix of bucket** pointers: $h_d(\text{key}) = n$, where n is the number of $[x_d, y_d)$ with $\text{key} \in [x_d, y_d)$ in the dimension d

Example (2 dimensions): Find record with $(a, b) = (22, 31)$

- Apply 'hashes' $h_a(22) = \#[20, 30)_a = 3$, $h_b(31) = \#[0, 100)_b = 4$
- Retrieve the pointer to the bucket for cell $(3, 4)$ from pointer matrix



Search: easy

Insertion & Deletion: inherit the problems of usual hashes

Bitmap indexes

Bitmap index:

- each value of each indexed attribute has a bitmap vector
- each tuple has a component in each vector

Efficient for several attributes with small domains

Take a lot of space, but can be compressed

Search: bit-wise AND

Insertion & Deletion: simple for existing values

File			
record-number	F	G	H
1	30	foo	65
2	30	bar	43
3	40	baz	84
4	50	fou	43
5	40	bar	65
6	30	baz	65

Bitmap vectors for F						
30:	1	1	0	0	0	1
40:	0	0	1	0	1	0
50:	0	0	0	1	0	0

Bitmap vectors for G						
bar:	0	1	0	0	1	0
baz:	0	0	1	0	0	1
foo:	1	0	0	0	0	0
fou:	0	0	0	1	0	0

Indexes:

The second key DBMS component for efficient query evaluation

IN3020&4020 – Database Systems (2024)

Part 1: Query Processing

Lectures 11, 12: Query Compilation and Optimisation

19, 20 February

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key **ingredients**:

Relational Algebra (✓) and Indexes (✓)

We **have studied** these ingredients (almost in isolation)
and **now** put all things together

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⎵ parsing

(parse tree)

⎵ compilation

$$\pi_{\text{title}}(\sigma_{\text{starName}=\text{name}}(\mathbf{StarIn} \times \pi_{\text{name}}(\sigma_{\text{birthDate LIKE '%1960'}}(\mathbf{MovieStar}))))$$

⎵ optimisation

$$\pi_{\text{title}}(\mathbf{StarIn} \bowtie_{\text{starName}=\text{name}} \pi_{\text{name}}(\sigma_{\text{birthDate LIKE '%1960'}}(\mathbf{MovieStar})))$$

⎵ evaluation over tables

(Result)

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⎵ parsing

(parse tree)

⎵ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarsIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$$

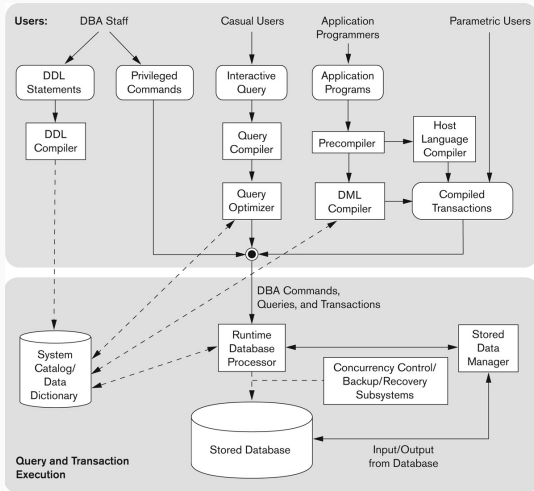
⎵ optimisation

$$\pi_{title}(\mathbf{StarsIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$$

⎵ evaluation over tables (next time)

(Result)

The place of parsing and optimisation



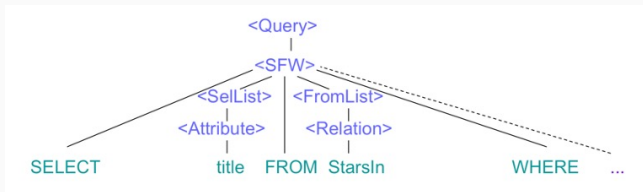
1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 18 (beginning) and 19
(my lectures have different accents)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 16
(closer to my presentation)
3. etc.

1. Parsing
2. Compilation
3. Optimisation: Overview
4. Optimisation: Cost Function
5. Optimisation: Heuristics

1. Parsing

Parsing: overview

Parsing transforms an SQL (`SELECT`) query to a *parse tree*—that is, from `SELECT title FROM StarsIn WHERE ...` to



Parse tree:

- **inner nodes** are *syntactic categories* for parts of the query
- **leaves** are *primitive atoms* representing keywords, names, constants, parentheses, operators, etc.

Parsing is done by standard techniques using a (*context-free*) *grammar* for SQL (see details in a *Compiler Book*)

Excerpt of SQL (DQL) grammar in BNF

`<Query> ::= <SFW> | "(" <Query> ")" | <Query> "UNION" <Query> | ...`

`<SFW> ::= "SELECT" (<SelList>|"*") "FROM" <FromList> ["WHERE" <Condition>]
["GROUP BY" ... ["HAVING" ...]] ["ORDER BY" ...]`

`<SelList> ::= <Attribute> | <Attribute> "," <SelList> | ...`

`<FromList> ::= <Relation> | <Relation> "," <FromList> | ...`

`<Condition> ::= <Condition> "AND" <Condition> | "NOT" <Cond> |
<Attribute> "IN" <Query> | <Attribute> "=" <Attribute> | ...`

...

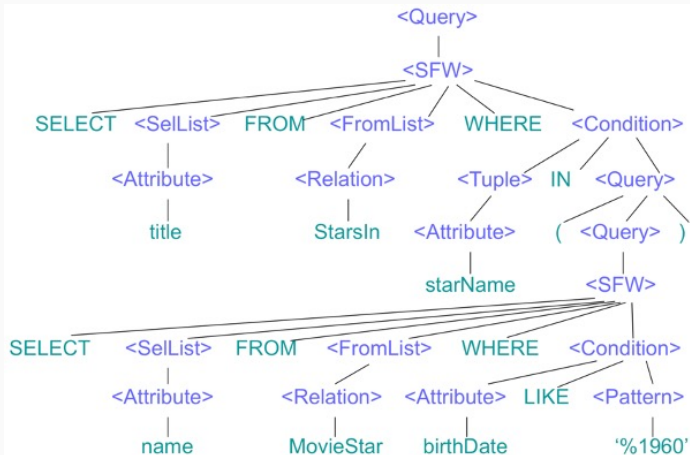
`<Attribute>, <Relation> ::= ... (parse to strings of symbols)`

...

Parsing: Example

```
SELECT title FROM StarsIn WHERE starName IN  
( SELECT name FROM MovieStar WHERE birthDate LIKE '%1960' );
```

parses to



Besides building the tree, several **checks** is done using the **System Catalog**:

- **resolving names**: checking that all the tables and attributes **exist** (and, if successful, linking them)
- **type checking**: all operations in expressions are **well-typed**

Both building the tree and the checks can **fail**

2. Compilation

Query compilation: Idea

Converts (i.e., 'compiles') the parse tree to an RA expression, called initial (logical) query plan:



Note 1: I use **tree-like representation** for the RA expression (natural and intuitive)

Note 2: I use the **relation names** instead of full relations (for brevity)

Query compilation: Subquery-free case

Compilation without subqueries is direct

For **example**, the parse tree for

```
SELECT Atts FROM Table1, ..., TableM WHERE C
```

with `Table1, ..., TableM` without common attributes compiles as

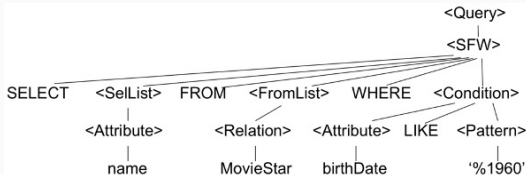
- **replace** `Table1, ..., TableM` with
 $(\dots (Table1 \times Table2) \times \dots) \times TableM$ in the tree
- apply selection $\sigma_C(\dots)$ to the product
- apply projection $\pi_{Atts}(\dots)$ to the selection

If `Table1, ..., TableM` have common attributes, we need renaming

Note: `INNER JOIN ... ON ...`, `OUTER JOIN ... ON ...` make two copies of join attributes; need care when compiling to \bowtie_C , \bowtie

Query compilation: Subquery-free example

```
SELECT name FROM MovieStar WHERE birthDate LIKE '%1960'
```



Compiling queries with subqueries

Recall: FROM clause may mention sub-SELECT

Recall: conditions in WHERE clause can also have subqueries:

```
SELECT ... WHERE ... EXISTS(SELECT ... ) ... ;
```

```
SELECT ... WHERE ... (Att1, ..., AttN) op SOME(SELECT ... ) ... ;
```

```
SELECT ... WHERE ... (Att1, ..., AttN) op ALL(SELECT ... ) ... ;
```

here, `op` is one of `=`, `>=`, `>`, ...

`SOME(...)` has synonym `ANY(...)`

`= SOME(...)` has synonym `IN(...)`

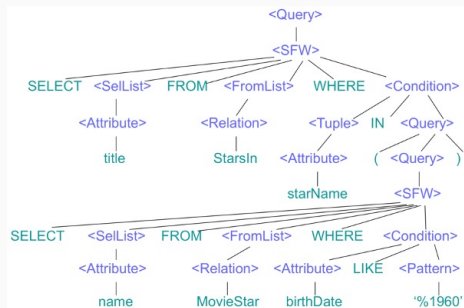
Can be **correlated**: subquery can use attributes from the outside

Compiling subqueries is more **challenging**,
especially with **correlated attributes** (but always possible)

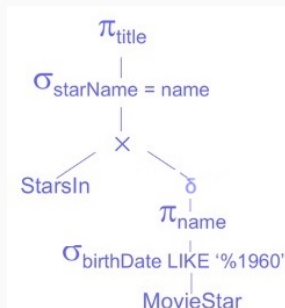
Non-correlated IN-subqueries

```
SELECT title FROM StarsIn WHERE starName IN  
( SELECT name FROM MovieStar WHERE birthDate LIKE '%1960' );
```

↴



↷



Alternatively:

$$\pi_{title}(StarsIn \bowtie (\pi_{name \rightarrow starName}(\sigma_{birthDate LIKE '%1960'}(MovieStar))))$$

Other non-correlated subqueries

Subqueries with NOT IN are **symmetric**:

```
SELECT title FROM StarsIn WHERE starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⋈

$$\pi_{\text{title}}(\text{StarsIn} \triangleright (\pi_{\text{name} \rightarrow \text{starName}}(\sigma_{\text{birthDate LIKE '%1960'}}(\text{MovieStar}))))$$

If a condition with a subquery is a **sub-condition** of a complex Boolean expression, it can be **isolated**:

```
SELECT title FROM StarsIn WHERE (starName NOT IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960')) AND Cond;
```

⋈

$$\pi_{\text{title}}(\sigma_{\text{Cond}}(\text{StarsIn}) \triangleright (\pi_{\text{name} \rightarrow \text{starName}}(\sigma_{\text{birthDate LIKE '%1960'}}(\text{MovieStar}))))$$

Other (non-correlated) sub-queries are **similar** (incl. subqueries in FROM)

Correlated subqueries by copying

For **correlated subqueries**, we may create a **copy** of the outer table in the subquery:

```
SELECT Person.name FROM Person
   WHERE EXISTS(SELECT * FROM Address
                WHERE Person.name = Address.name);
```

↴

$$\pi_{Person.name}(Person \times (\sigma_{Person.name=Address.name}(Person \times Address)))$$

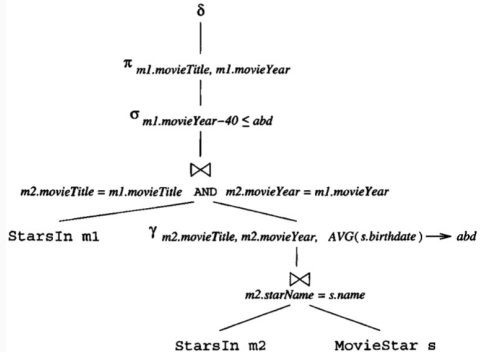
Correlated subqueries by postponing selection

Alternatively, we may **postpone** the selection

Example: Movies with average age of starring actors at most 40

```
SELECT DISTINCT m1.movieTitle, m1.movieYear
FROM StarsIn m1
WHERE m1.movieYear - 40 <= ANY (
  SELECT AVG(birthdate)
  FROM StarsIn m2, MovieStar s
  WHERE m2.starName = s.name AND
        m1.movieTitle = m2.movieTitle AND
        m1.movieYear = m2.movieYear
);
```

→



3. Optimisation: Overview

Optimisation: Idealistic goal

Given a query plan (RA expression) $Expr$, our **idealistic** goal is to find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ give the **same answer** over the given tables
- $Expr'$ can be evaluated as **quick as possible**
- finding $Expr'$ and **ensuring** the first two conditions is also quick

There are few **serious** (in fact, usually **crucial**) obstacles

- search space for $Expr'$ is huge (actually, infinite)
- in general, the **only way to guarantee** that some $Expr'$ can be evaluated quicker than some other $Expr$ is to **evaluate both** and measure time
- **by the way**, without the last condition, $Expr'$ is obvious: expression **constructing** the result of $Expr$ (ignoring relations)

Optimisation: Pragmatic approach

Given a (logical) query plan (RA expression) $Expr$, DBMSs **pragmatically** find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ are **equivalent** (i.e., same answer for **all tables**)
- $Expr'$ is **estimated** to be **quicker** than other candidates
(for the **given tables**)
- finding candidate $Expr'$ and **estimation** is quick

This approach is based on **two components**:

1. good **cost function** f for query plans that is quick to compute and **reflects** evaluation time (i.e., $f(Expr') \leq f(Expr)$ in most cases means that $Expr'$ is quicker to evaluate than $Expr$)
2. good **heuristic** (usually based on **RA algebraic laws**) that allows to quickly construct (one or several) candidate $Expr'$ with (much) better cost

Optimisation: Example

SELECT B, C, Y FROM R, S WHERE W=X AND A=3 AND Z='a'

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Candidate 1: Initial query plan

SELECT B, C, Y FROM R, S WHERE W=X AND A=3 AND Z='a'
compiles to initial plan

$\pi_{B,C,Y}(\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

\times
 \rightsquigarrow

A	B	C	...	W	X	Y	Z
1	z	1	...	4	1	a	a
...	2	f	c
...	2	f	c
2	c	6	...	2	1	a	a
...	2	f	c
...
3	r	8	...	7	1	a	a
...	2	f	c
...
3	r	8	...	7	7	k	a
4	n	9	...	4	1	a	a
...	2	f	c
...	v
2	i	0	...	3	1	a	a
...	2	c	c
...
3	t	5	...	9	1	a	a
...	2	f	c
...
7	e	3	...	3	1	a	a
...	2	f	c
...	v

σ, π
 \rightsquigarrow

B	C	Y
r	8	k

Intuitively, cannot be the most efficient

Candidate 2: Pushing selection down

Initial plan $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$
transforms, by **pushing selection** and **theta-join definition** laws, to
another plan $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, σ
 \rightsquigarrow

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\bowtie_{W=X}$
 \rightsquigarrow

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 \rightsquigarrow

B	C	Y
r	8	k

Looks **better**, especially, if we use **indexes** on **A** in **R** and **Z** in **S**

Candidate 3: More fine-grained

Candidate 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X} \ \& \ A=3 \ \& \ Z='a')(R \times S)$

Candidate 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \ \bowtie_{W=X} \ \sigma_{Z='a'}(S)))$

Candidate 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \ \bowtie_{W=X} \ \sigma_{Z='a'}(S \times_{W=X} \sigma_{A=3}(R))))$

(here, $\times_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and \times)

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, \times
 \rightsquigarrow

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie_{W=X}$
 \rightsquigarrow

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 \rightsquigarrow

B	C	Y
r	8	k

More operations, but **quicker**

especially, if we use **indexes** on A in R and X (not $Z!$) in S

4. Optimisation: Cost function

Cost function requirements

Recall: We need **cost function** f such that

$f(\text{Expr}') \leq f(\text{Expr})$ in most cases means that

plan Expr' is **quicker** to evaluate than plan Expr

Evaluation time is spent for

- **reading** and **writing** to the secondary memory (e.g., disk I/O)
- **evaluating** RA operations in the main memory

Real DBMS (advanced) cost functions **take into account**:

- (potentially different) **evaluation time** for operations (using **algorithms** discussed in the next lecture)
- **indexes** that decrease the number of reads (**we know**)
- **reusing** intermediate results (e.g., as in Candidate 3 above)
- **pipelining** intermediate results (tuples) from one operation to another without writing back to the disk

Our cost function

We take as our cost function:

the **sum** of **estimates** of **sizes** (in bytes) of the **intermediate** results (i.e., subexpressions) of the plan (i.e., the RA expression)

- **estimate**, because exact may be difficult to compute (our **estimate** is computed as described below)
- **intermediate**, because the inputs and the outputs are the same
- **reflects** the most important thing above: the number of disk I/O (number of used blocks is roughly proportional to the size of a relation)
- **compositional**: knowing estimates for immediate sub-plans, we can quickly compute the estimate for the plan (e.g., if we know the size estimates of **Expr** and **Expr'**, we can estimate the size of **Expr** \bowtie **Expr'**)
- **quick** to compute

Essentially, we assume that each operation is executed in negligible time and in isolation, and all the time is spent in reading all arguments (in whole) of each operation from the disk and writing the result back

Before we go: Example

Numbers are **estimates**(!) of subquery sizes

(e.g., 16000 may not be the real size of the overall answer, may be **different** on two sides, and **not counted** anyway)



Our cost of the left plan: $2,000+400+32,000 = 34,400$

Our cost of the right plan: $2,000+400+16,000,000+32,000 = 16,034,400$

Reflects our intuition: **good cost function** (at least for this case)

Thus, our **main** metric, for a (base or subquery) relation R :

$Size(R)$ – (estimation of) the size of R in bytes

(i.e., the main component of our **cost**, which is the sum of sizes)

To estimate $Size(R)$ by **induction** by the query structure,
we will also maintain **secondary** metrics:

$Tups(R)$ – (estimation of) the **number of tuples** in relation R

$TupSize(R)$ – the (average) **size of a tuple** in R

as a result, we have $Size(R) = Tups(R) * TupSize(R)$

$Vals(R.A)$ – (estimation of) the number of **different values** of
attribute A in R

real DBMSs use **histograms** instead (we will discuss them later)

Metrics for the base case: Statistics of physical relations

To compute costs for complex plans by **induction**, we need to start with **statistics**—that is, $Size(R)$, $Tups(R)$, $TupSize(R)$, $Vals(R.A)$, etc., for the base database relations R

Computing statistics may be **expensive**, and doing it after each update may be **prohibitively costly**

However, we do **not need the exact** values every time we estimate the cost of a plan, some **inaccuracies** is not a problem

System can recompute statistics only **time to time**
(automatically or manually)

Metrics for inductive case: Bag projection

Assuming **uniform** size of tuples (i.e., no VARCHAR, etc.), the size and almost all secondary metrics for **bag (generalised) projection** $\pi_L^b(R)$ can be calculated accurately from metrics for R , (due to **bag semantics**, one result tuple for each argument tuple):

- $Tups(\pi_L^b(R)) = Tups(R)$
- $TupSize(\pi_L^b(R))$ is defined by L
- $Size(\pi_L^b(R)) = Tups(\pi_L^b(R)) * TupSize(\pi_L^b(R))$
- $Vals((\pi_L^b(R)).A) = Vals(R.A)$ if A is an attribute in R
- $Vals((\pi_{Expr \rightarrow A}^b(R)).A)$ depends on $Expr$;
by default, can be estimated as
 $\max(Vals(R.A_1), \dots, Vals(R.A_n))$ where A_1, \dots, A_n are all different attributes of R mentioned in $Expr$

(We will cover **set projection** with duplicate elimination)

Metrics for bag projection: Example

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = 4 + 20 + 4 + 10 = 38$
(INT(4) + CHAR(20) + INT(4) + CHAR(10))
- $Size(R) = 5 * 38 = 190$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

$S = \pi_{A+10 \rightarrow Aplus, B, A+C \rightarrow AC}^b(R)$:

- $Tups(S) = 5$
- $TupSize(S) = 4 + 20 + 4 = 28$
- $Size(S) = 5 * 28 = 140$
- $Vals(S.Aplus) = 5$, $Vals(S.B) = 2$ (both exact),
 $Vals(S.AC) = 5$ (estimation, 4 in reality)

Metrics for inductive case: Selection (overview)

The size of selection $\sigma_{Cond}(R)$ is much more difficult to estimate:

- $TupSize(\sigma_{Cond}(R)) = TupSize(R)$ (easy)
- so, $Size(\sigma_{Cond}(R)) = Tups(\sigma_{Cond}(R)) * TupSize(\sigma_{Cond}(R))$
relies on estimation of $Tups(\sigma_{Cond}(R))$
- we consider different types of *Cond* separately
- by default, we naively assume uniform and independent distribution of $Vals(R.A)$ across relevant $R.A$
- real DMBSs use histograms for much better estimations
- estimation of all $Vals((\sigma_{Cond}(R)).A)$ with A mentioned in *Cond* can be done similarly to *Tups* (try it by yourself);
for other attributes A we can take
$$Vals((\sigma_{Cond}(R)).A) = Vals(R.A) * Tups(\sigma_{Cond}(R)) / Tups(R)$$

Metrics for inductive case: Selection (atomic comparisons)

Estimating $Tups(\sigma_{Cond}(R))$ for atomic $Cond$:

- equality to a constant:

$$Tups(\sigma_{A=c}(R)) = Tups(R) / Vals(R.A) \text{ (may be not integer!)}$$

- inequality to a constant:

$$Tups(\sigma_{A \neq c}(R)) = Tups(R) * (1 - 1 / Vals(R.A)) \text{ (or just } Tups(R))$$

- comparison to a constant:

$$Tups(\sigma_{A < c}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

- equality of attributes:

$$Tups(\sigma_{A=B}(R)) = (Tups(R) / \max(Vals(R.A), Vals(R.B)))$$

- inequality of attributes:

$$Tups(\sigma_{A \neq B}(R)) = Tups(R) * (1 - 1 / \max(Vals(R.A), Vals(R.B))) \text{ (or } \dots)$$

- comparison of attributes:

$$Tups(\sigma_{A < B}(R)) = Tups(R) / 3 \text{ (or just } Tups(R) / 2)$$

- comparisons with functions:

$$Tups(\sigma_{f(A_1, \dots, A_n) \text{ op } g(B_1, \dots, B_n)}(R)) = \dots$$

Metrics for atomic selection: Examples

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = \dots$
- $Size(R) = \dots$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

Estimates of the number of tuples:

- $Tups(\sigma_{A=4}(R)) = 5/5 = 1$ (real: 1),
 $Tups(\sigma_{A \neq 4}(R)) = 5 * (1 - 1/5) = 4$ (4),
 $Tups(\sigma_{A < 4}(R)) = 5/3 = 1.66$ (3)
- $Tups(\sigma_{B='cat'}(R)) = 5/2 = 2.5$ (3),
 $Tups(\sigma_{B \neq 'cat'}(R)) = 5 * (1 - 1/2) = 2.5$ (2)
- $Tups(\sigma_{A=C}(R)) = 5/5 = 1$ (0),
 $Tups(\sigma_{A \neq C}(R)) = 5 * (1 - 1/5) = 4$ (5),
 $Tups(\sigma_{A < C}(R)) = 5/3 = 1.66$ (5)

Metrics for inductive case: Selection (Boolean combinations)

Estimating $Tups(\sigma_{Cond}(R))$ for Boolean combinations of conditions:

- **AND:** $Tups(\sigma_{Cond_1 \& Cond_2}(R)) = Tups(\sigma_{Cond_1}(\sigma_{Cond_2}(R))) = Tups(\sigma_{Cond_2}(\sigma_{Cond_1}(R))) = Tups(R) * factor_{Cond_1} * factor_{Cond_2}$
(assumes independence)
- **NOT:** $Tups(\sigma_{\neg Cond'}(R)) = Tups(R) - Tups(\sigma_{Cond'}(R))$
- **OR:** $Tups(\sigma_{Cond_1 \vee Cond_2})(R) = Tups(\sigma_{\neg(\neg Cond_1 \& \neg Cond_2)}(R)) = Tups(R) * (1 - (1 - factor_{Cond_1}) * (1 - factor_{Cond_2}))$
(assumes independence)

Metrics for selections with Boolean combinations: Examples

Relation R :

A	B	C	D
1	cat	1999	a
2	cat	2002	b
3	dog	2002	c
4	cat	1998	a
5	dog	2000	c

Statistics of R :

- $Tups(R) = 5$
- $TupSize(R) = \dots$
- $Size(R) = \dots$
- $Vals(R.A) = 5$, $Vals(R.B) = 2$,
 $Vals(R.C) = 4$, $Vals(R.D) = 3$

Estimates of the number of tuples:

- $Tups(\sigma_{A=4 \ \& \ B='cat'}(R)) = 5 * 1/5 * 1/2 = 0.5$ (real: 1),
 $Tups(\sigma_{A \neq 4 \ \& \ B \neq 'cat'}(R)) = 5 * 4/5 * 1/2 = 2$ (2)
- $Tups(\sigma_{A=4 \ \vee \ B='cat'}(R)) = 5 * (1 - (1 - 1/5) * (1 - 1/2)) = 3$ (3),
 $Tups(\sigma_{A \neq 4 \ \vee \ B \neq 'cat'}(R)) = 5 * (1 - (1 - 4/5) * (1 - 1/2)) = 4.5$ (4)

Metrics for inductive case: Natural join (overview)

The size of **natural join** $R \bowtie S$ (and other joins) is also difficult:

- $TupSize(R \bowtie S)$ is the sum of sizes of distinct attributes in R and S
- so, $Size(R \bowtie S) = Tups(R \bowtie S) * TupSize(R \bowtie S)$ relies on estimation of $Tups(R \bowtie S)$
- to estimate $Tups(R \bowtie S)$ and $Vals((R \bowtie S).A)$ for join attributes A we rely on **value containment** assumption:
If A is an attribute in R and S such that $Vals(R.A) \leq Vals(S.A)$ then **every value** of A in R **appears** in A of S (cf. **foreign key**)
- to estimate $Vals((R \bowtie S).A)$ for non-join attributes A we rely on **value preservation** assumption:
If A is in R but not in S then $Vals((R \bowtie S).A) = Vals(R.A)$ (and symmetric for A in S but not in R)
- real DMBSs use **histograms** for much better estimations
- **other joins** are expressible, and we can use **the defining laws** for estimation

Metrics for inductive case: Natural join (one join attribute)

Estimating $R \bowtie S$ with **one** join (i.e., common) attribute A
(using these two assumptions):

- if $Vals(R.A) \leq Vals(S.A)$ then each tuple in R will match approximately $Tups(S)/Vals(S.A)$ tuples in S
- thus, in general, $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.A), Vals(S.A))$
- $Vals((R \bowtie S).A) = \min(Vals(R.A), Vals(S.A))$
- $Vals((R \bowtie S).A') = Vals(R.A')$ for each non-join A' in R
- $Vals((R \bowtie S).A') = Vals(S.A')$ for each non-join A' in S

Metrics for natural join: Example 1

Relations R and S :

Relation R

A	B	C	...	X
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
5	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Statistics of R and S :

- $Tups(R) = 9, Tups(S) = 9$
- $TupSize(R) = \dots, TupSize(S) = \dots$
- $Size(R) = \dots, Size(S) = \dots$
- $Vals(R.A) = 6, Vals(R.X) = 7,$
 $Vals(S.X) = 9, Vals(S.Z) = 3$

Estimates (all correct, since essentially primary & foreign keys):

- $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.X), Vals(S.X)) = 9$
(real: 9)
- $Vals((R \bowtie S).X) = \min(Vals(R.X), Vals(S.X)) = 7$ (7)
- $Vals((R \bowtie S).A) = Vals(R.A) = 6$ (6)
- $Vals((R \bowtie S).Z) = Vals(S.Z) = 3$ (3) (these may be different even with the keys)

Metrics for natural join: Example 2

Relations R and S :

Relation R

A	B	C	...	X
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	0

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
5	k	a
6	e	a
7	g	c
8	i	b
9	e	c

Statistics of R and S :

- $Tups(R) = 9, Tups(S) = 9$
- $TupSize(R) = \dots, TupSize(S) = \dots$
- $Size(R) = \dots, Size(S) = \dots$
- $Vals(R.A) = 6, Vals(R.X) = 7,$
 $Vals(S.X) = 9, Vals(S.Z) = 3$

Estimates (no foreign key any more):

- $Tups(R \bowtie S) = Tups(R) * Tups(S) / \max(Vals(R.X), Vals(S.X)) = 9$
(real: 8)
- $Vals((R \bowtie S).X) = \min(Vals(R.X), Vals(S.X)) = 7$ (6)
- $Vals((R \bowtie S).A) = Vals(R.A) = 6$ (6)
- $Vals((R \bowtie S).Z) = Vals(S.Z) = 3$ (3)

Metrics for inductive case: Natural join (many join attributes)

Estimating $R \bowtie S$ with **several** join attributes A_1, \dots, A_n
(using these two assumptions):

- applying the same logic, we have $Tups(R \bowtie S) =$

$$\frac{Tups(R) * Tups(S)}{\max(Vals(R.A_1), Vals(S.A_1)) * \dots * \max(Vals(R.A_n), Vals(S.A_n))}$$

- for $n = 0$ we have **Cartesian product**:

$$Tups(R \bowtie S) = Tups(R) * Tups(S)$$

- $Vals((R \bowtie S).A)$ is as before:

$\min(Vals(R.A_i), Vals(S.A_i))$ for each join A_i ,

$Vals((R \bowtie S).A') = Vals(R.A')$ for non-join A' (in R)

Metrics for inductive case: bag-theoretic operations

Estimating metrics for bag union, intersection and difference:

- $TopSize(R \cup^b S)$, $TopSize(R \cap S)$, $TopSize(R \setminus S)$ are **inherited** from the arguments
- $Tups(R \cup^b S) = Tups(R) + Tups(S)$ (always),
 $Vals((R \cup^b S).A) = \max(Vals(R.A), Vals(S.A)) + \min(Vals(R.A), Vals(S.A))/2$ (a possibility for estimation)
- $Tups(R \cap S) = \min(Tups(R), Tups(S))/2$,
 $Vals((R \cap S).A) = \min(Vals(R.A), Vals(S.A))/2$
(possible estimates)
- $Tups(R \setminus S) = Tups(R) - Tups(S)/2$,
 $Vals((R \setminus S).A) = Vals(R.A) - Vals(S.A)/2$
(possible estimates)
- set-theoretic union can be estimated via **duplicate elimination** (see below)

Metrics for inductive case: duplicate elimination and grouping

Estimating metrics for **duplicate elimination**:

- $TupSize(\delta(R))$ and all $Vals((\delta(R)).A)$ are **inherited** from R
- $Tups(\delta(R))$ is somewhere in between 1 and $Tups(R)$, so we can estimate as $Tups(R)/2$
- however, $Tups(\delta(R))$ is also bounded by $Vals(R.A_1) * \dots * Vals(R.A_n)$ for A_1, \dots, A_n all attributes of R
- so, we can estimate $Tups(\delta(R)) = \min(Tups(R)/2, Vals(R.A_1) * \dots * Vals(R.A_n))$

Estimating metrics for **grouping with aggregation**:

- similar to duplicate elimination (write it down by yourself)
- the only new case is $Vals((\gamma_{Agg(A) \rightarrow A'}(R)).A')$:
we can take $Tups(\gamma_{Agg(A) \rightarrow A'}(R))/2$ (i.e., every aggregate value appears **twice** on average)

Histograms

We maintained the number $Vals(R.A)$ of distinct values of each attribute A of each R to estimate the size of selection and joins

Assumed **uniform** distribution of these values across the tuples in R

Real DBMSs uses more fine-grained **histograms**:

- **simple histogram**: number of tuples in R for each value of A
- **advanced** (when $Vals(R.A)$ is large): number of tuples in R for intervals of values of A (many techniques to pick the intervals)
- **combination** is also possible

Histograms allow us to compute the metrics (incl. $Size(R)$) more accurately

However, they are difficult to propagate estimations of histograms for complex RA expressions (i.e., not in the the statistics of tables)

Often, a **hybrid** approach is used: histograms in the statistics, $Vals(R.A)$ with uniform distribution assumption for sub-plans

Plan costs: Come back to the example

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

σ, \bowtie
 \rightsquigarrow

Relation R

A	B	C	...	W
1	z	1	...	4
2	c	6	...	2
3	r	8	...	7
4	n	9	...	4
2	j	0	...	3
3	t	5	...	9
7	e	3	...	3
8	f	5	...	8
1	h	7	...	5

Relation S

X	Y	Z
1	a	a
2	f	c
3	t	b
4	b	b
7	k	a
6	e	a
7	g	c
8	i	b
9	e	c

$\sigma, \bowtie_{W=X}$
 \rightsquigarrow

A	B	C	...	W	X	Y	Z
3	r	8	...	7	7	k	a

π
 \rightsquigarrow

B	C	Y
r	8	k

Plan 1 (initial): $\pi_{B,C,Y} (\sigma_{W=X \ \& \ A=3 \ \& \ Z='a'}(R \times S))$

Plan 2: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S)))$

Plan 3: $\pi_{B,C,Y} ((\sigma_{A=3}(R) \bowtie_{W=X} \sigma_{Z='a'}(S \bowtie_{W=X} \sigma_{A=3}(R))))$

(here, $\bowtie_{W=X}$ is 'equi-semi-join', naturally expressible via ρ and \bowtie)

Cost of Plan 2 ($Size(\sigma_{A=3}(R)) + Size(\sigma_{Z='a'}(S)) + Size(\dots \bowtie_{W=X} \dots)$) is **clearly better** than cost of Plan 1 ($Size(R \times S) + Size(\sigma_{W=X, A=3, Z='a'}(R \times S))$)

Homework: compute and compare the costs of Plans 2 and 3

5. Optimisation: Heuristics

Searching for a plan with good cost

As we discussed, given an initial query plan (RA expression) $Expr$, we have a huge search space of plans to compare with

DBMSs **pragmatically** find **another** query plan $Expr'$ such that

- $Expr$ and $Expr'$ are equivalent (i.e., give the **same answer** on all possible data instances)
- $Expr'$ is **estimated** to be **quicker** than other candidates using the cost function (for us: sum of sizes of intermediate sub-queries in the plan)

They use **greedy heuristic**:

- starting from the initial $Expr$, they apply the algebraic laws one by one, trying to **decrease** the cost on (almost) each step as much as possible
- stops with a selected plan when there are no more cost improvements

The **algebraic laws** that (almost always) decrease our cost:

- **Push selections** as far down as possible
- If the selection condition consists of several parts (**AND** or **OR**), **split into multiple selections** and **push** each one as far down the tree as possible
- **Push projections** as far down as possible
- **Combine selections and joins** (especially, Cartesian products) to appropriate theta-joins
- Apply **idempotence** as much as possible

Real DBMS cost functions are more fine-grained, and these rules may be less universal (e.g., due to indexes)

Order of joins

One of the most important practical question for optimisation is the order to join—that is, how to evaluate

$$(\dots (R_1 \bowtie R_2) \bowtie \dots \bowtie R_{n-1}) \bowtie R_n$$

Good estimates of intermediate results here are **crucial** for efficient evaluation (and the difference may be huge)

In general, the search space (all possible orders of ()) is large

Luckily, such joins are usually **not arbitrary**: the **join graph** has **bounded (hyper-)tree width** (**check it out!**)

Following the structure of the corresponding **hyper-tree** gives us an efficient join order (good cost)

Even the choice between $R_1 \bowtie R_2$ and $R_2 \bowtie R_1$ is **not trivial**: the join algorithms are **not symmetric** (but our cost function does not distinguish these cases)

What we learned?

How DBMSs **compile** queries to RA plans and **optimise** to presumably better plans

Next time: how plans are evaluated

IN3020&4020 – Database Systems (2024)

Part 1: Query Processing

Lectures 13, 14, 15: Query Evaluation

27 February, 4, 5 March

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases

Part 2. Transaction management in relational databases

Part 3. NoSQL DBMS

SQL query processing relies on two key **ingredients**:

Relational Algebra (✓) and Indexes (✓)

We **have studied** these ingredients (almost in isolation)
and **now** put all things together

SQL (DQL) query processor in action

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
(SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⌋ parsing

(parse tree)

⌋ compilation

$\pi_{title}(\sigma_{starName=name}(\mathbf{StarIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$

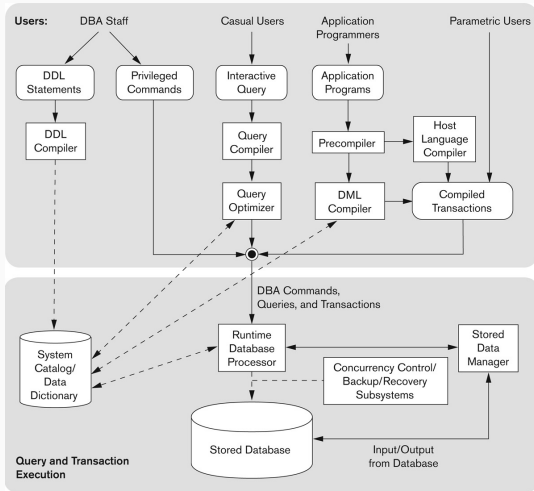
⌋ optimisation

$\pi_{title}(\mathbf{StarIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$

⌋ evaluation over tables

(Result)

The place of parsing and optimisation



1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition): Chapter 18
(my lectures have different accents)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Section 15
(much closer to my presentation)
3. etc.

1. Overview
2. Executing Basic Operations
3. Executing RA Operations
 - 3.1. Unary Operations
 - 3.2. Binary Operations
4. Summary
5. Query Plans in Real DBMSs

1. Overview

Physical Plans

Logical query plan: RA expression

Physical query plan: Logical query plan where an **algorithm** is assigned to each operation

In principle, we may have a choice between **several algorithms** for an operation with different incomparable characteristics (e.g., time consumption) and requirements (e.g., necessary memory, presence of indexes, etc.)

Moreover, algorithms may be **pipelined** so that characteristics of two algorithms together may be better than the sum of their characteristics in isolation

We used the **size of the intermediate relations** as costs of logical query plans, which are an approximation of time consumption of the algorithms

More fine-grained costs take the concrete algorithms into account, and estimate different **physical plans** rather than just **logical plans**

For simplicity, in what follows we ignore this issue and assume that we have a **logical plan** constructed

We will only study algorithms '**in isolation**', sometimes several for one operation (depends on available memory, indexes, etc.)

Before algorithms for the **RA operations**, we will consider algorithms for **basic operations** (sorting, hashing, etc.)

- need them by themselves, e.g., as intermediate steps
- they are often the base for RA algorithms

Algorithms for operations: Assumptions and costs

Our (new) **main cost**: number of **block I/O** (reads and writes)

For **each algorithm**, we generally assume

- the operands are initially **on the disk**
- the result is written to **output stream** (with no cost)
- other costs **can be ignored**

- the tuples of a relation are **clustered**—that is, **tightly packed in blocks** (no free space in a block for a new tuple, maybe spanned or unspanned)

- when we use indexes, we **ignore** the cost of their retrieval

We use the following parameters:

$MMBuffers$ – the size of the main memory, in blocks
(sometimes called **buffers**)

$Blocks(R)$ – the number of blocks to store relation R on disk

$Tups(R)$ – the **number of tuples** in relation R

$Vals(R.A)$ – the number of **different values** of (one or several) attributes A in R

($Tups(R)$ and $Vals(R.A)$ are now exact, not estimation)

2. Executing basic operations

Scanning a relation is the most simple: just to retrieve the whole relation

- **Table scan:** no index used, blocks retrieved one by one
Cost: $Blocks(R)$ disk I/O operations
(If clustered; otherwise, up to $Tups(R)$, or even more)
- **Index scan:** Index (sparse or dense) is used
Cost: $Blocks(R)$ disk I/O operations (the same)
(But may be used to retrieve a part of R quicker)

Sorting a relation

Sort scan reads a relation R and returns it in sorted order with respect to a set of attributes $Atts$

- if R has an index of $Atts$, then it can be used to traverse the tuples in R in the order
 - Cost: $Blocks(R)$ (primary/clustering) or $Tups(R)$ (secondary) disk I/O operations
- if $Blocks(R) \leq MMBuffers$ (i.e., the whole R fits into the main memory), we use an efficient sorting algorithm
 - Cost: $Blocks(R)$ disk I/O operations
- if $Blocks(R) > MMBuffers$ (i.e., R is too big to fit into memory), we use a dedicated sorting algorithm that handles data in multiple passes
 - we consider two-phase multiway merge sort (TPMMS)
 - (has requirement $\sqrt{Blocks(R)} \leq MMBuffers$,
 - but can be generalised to N-phase multiway merge sort)

Two-Phase Multiway Merge Sort (TPMMS): Phase 1

TPMMS Phase 1. Sort parts of R of size $MMBuffers$ individually:

Repeat:

1. Fill all ($MMBuffers$) buffers with new blocks of R
2. Sort this part in the memory
3. Write the sorted result back to disk

Until all R is processed

Cost of Phase 1: $2 * Blocks(R)$

Two-Phase Multiway Merge Sort (TPMMS): Phase 2

TPMMS Phase 2. Merge all the sorted parts into one:

Repeat:

1. Read the first block from each part into buffers (at most $MMBuffers$)
2. Find the smallest tuple over the first tuples in the buffers
3. Remove this tuple from the buffer; if the buffer is empty, re-fill it with the next block from the same part
4. Put this tuple to the output buffer; if it is full, flush it

Until all parts in the main memory is empty

Cost of Phase 2: $Blocks(R)$

Total cost: $3 * Blocks(R)$

Requirement: Number of parts cannot be more than $MMBuffers - 1$:

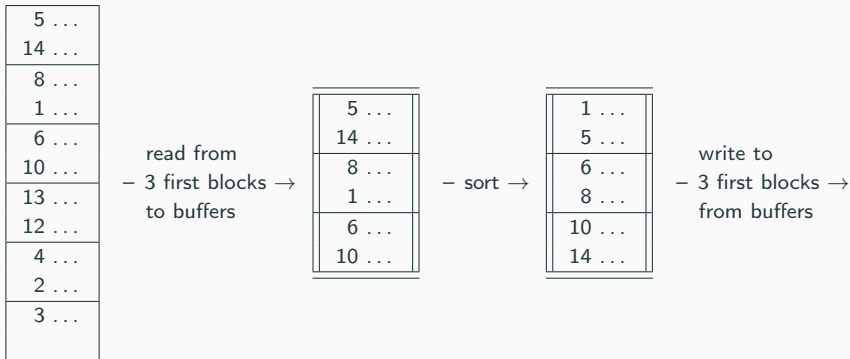
$$Blocks(R)/MMBuffers \leq MMBuffers - 1$$

$$\text{so essentially } \sqrt{Blocks(R)} \leq MMBuffers$$

TPMMS: Example Phase 1

Let $MMBuffers = 3$

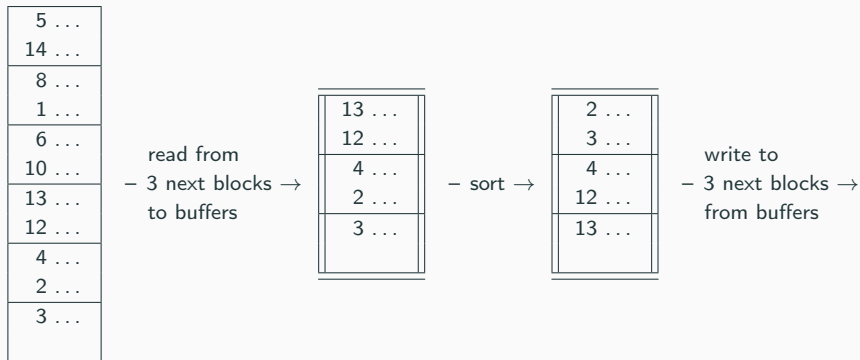
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 1

Let $MMBuffers = 3$

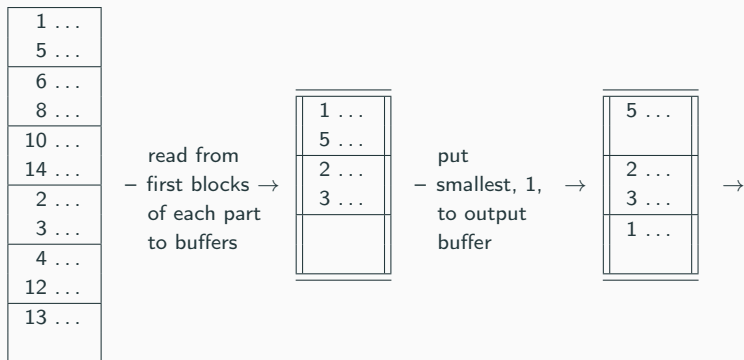
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

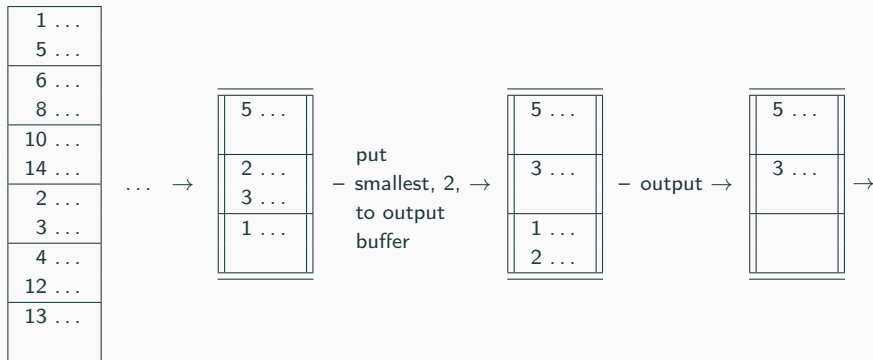
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

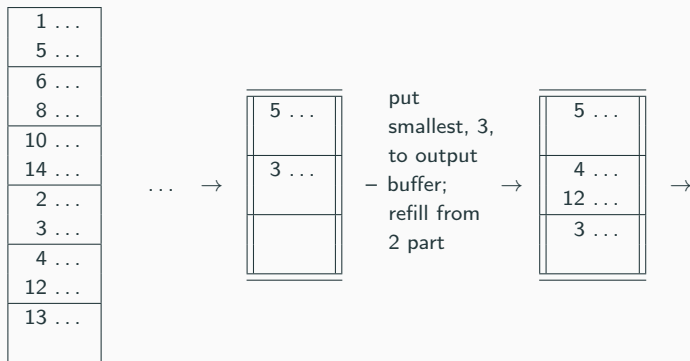
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

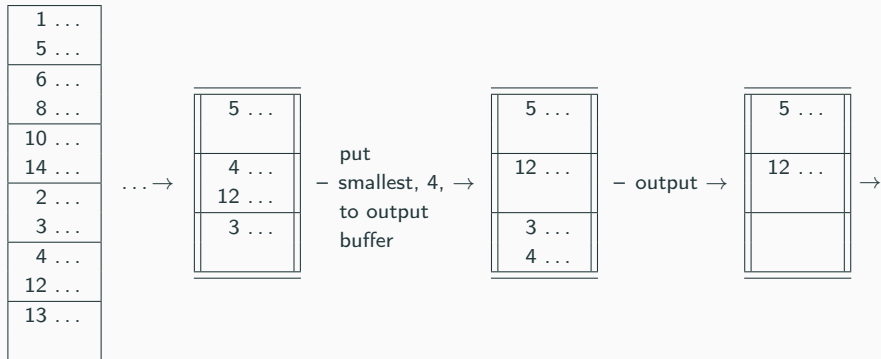
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

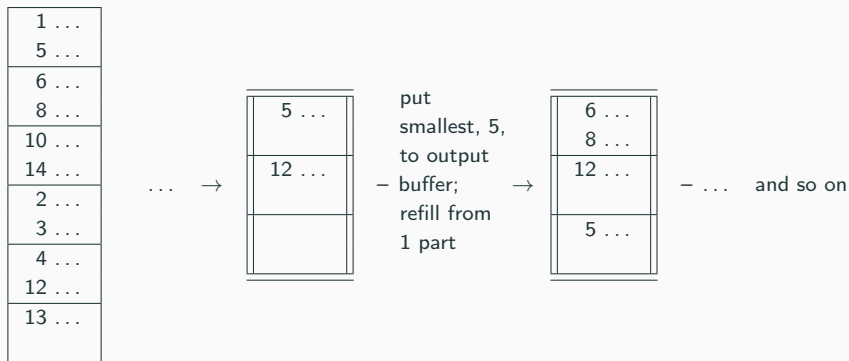
Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



TPMMS: Example Phase 2

Let $MMBuffers = 3$

Consider $Tups = 11$ tuples in $Blocks = 6$ (2 tuples per block):



Hash Partitioning

Partitions relation R to buckets according to hash function h
(applied to one or more attributes of R)

Condition: the number of hash values (i.e., buckets) is no more than $MMBuffers - 1$

Uses one **read** buffer and (at most) $MMBuffers - 1$ buffers associated to the hash values

Repeat:

1. If the read buffer is empty,
load the next block of R to the read buffer
2. For each tuple t in the read buffer
 - add t to the bucket buffer $h(t)$
 - if the bucket buffer is full, flush it to the output

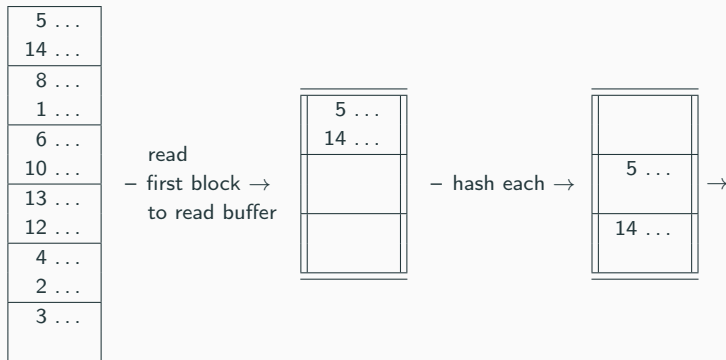
Until all blocks of R are processed

Cost: $Blocks(R)$ (or up to $Tups(R)$ if not clustered)

Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod 2$

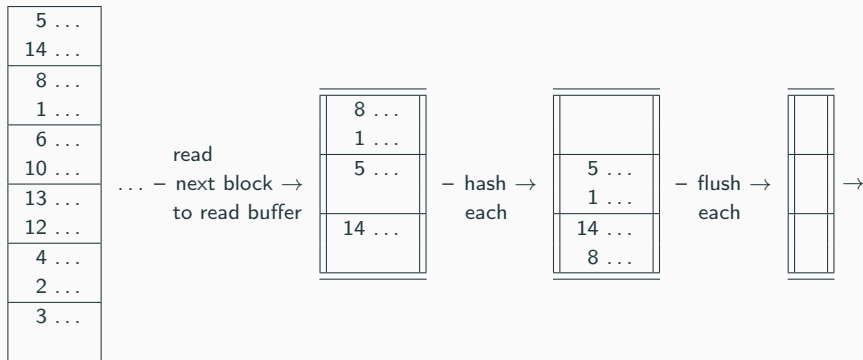
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod 2$

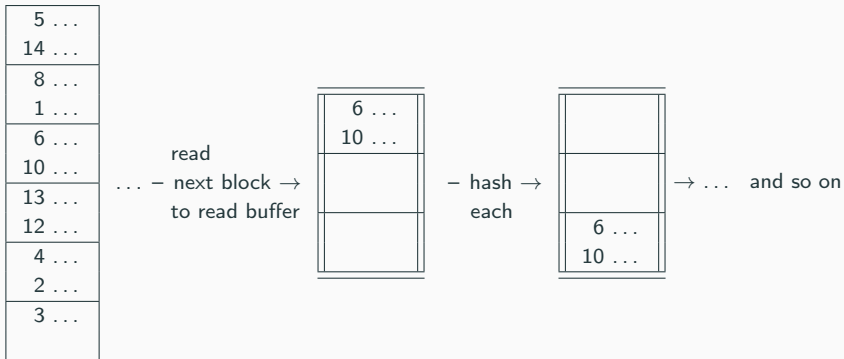
Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



Hash Partitioning: Example

Let $MMBuffers = 3$ and $h(t)$ is $Mod 2$

Consider $Tups = 11$ in $Blocks = 6$ (2 tuples per block):



3. Executing RA Operations

Classes based on **approach**:

- Sort-based (always work)
- Hash-based (relies on a good hash function)
- Index-based (relies on indexes)

Classes based on memory **requirements**:

- One-pass: data **fits** into the main memory,
read once
- Two-pass: data **does not** fit into the memory,
but **two passes** are enough
- N -pass, $N \geq 3$: recursive **generalisation** to more memory,
several passes (mostly omitted below)

3.1. Unary operations

Tuple-at-a-time operations: Selection and Projection

Selection $\sigma_{Cond}(R)$ and projection $\pi_L(R)$ process each tuple of R in isolation, so they have simple algorithms, regardless of $Blocks(R)$:

Repeat:

1. Retrieve the next relevant block into a buffer
(use indexes if available and relevant)
2. Process tuples in the buffer one by one, output when relevant

Until all relevant blocks of R are processed

Cost (I/O reads):

- $Blocks(R)$ for selection without index and projection
- for selection with index, depends on condition and index type
for example, $\lceil Blocks(R)/Vals(R.A) \rceil$ on average
for $A = c$ as $Cond$ and primary/clustering index

Requirement: (just) $1 \leq MMBuffers$

Selection with secondary index: Example

Let $Tups(R) = 20000$, $Blocks(R) = 1000$

Cost of $\sigma_{A=c}$ with index on A :

- for $Vals(R.A) = 10$ we have $1000/10 = 100$ disk I/O
- for $Vals(R.A) = 100$ we have $1000/100 = 10$ disk I/O
- for $Vals(R.A) = 20000$ (i.e., candidate key) we have
 $\lceil 1000/20000 \rceil = 1$ disk I/O

One-pass duplicate elimination

Duplicate elimination $\delta(R)$

when it can be performed in main memory

Use one read buffer and other buffers to store the result

Repeat:

1. Retrieve the next block of R in the read buffer
2. For each tuple t in the read buffer,
search for the same tuple in the result buffers,
append if not found

Until all blocks of R are processed

Cost (I/O reads): $Blocks(R)$

Requirement: $Blocks(\delta(R)) \leq MMBuffers - 1$

Two-pass sort-based duplicate elimination

Duplicate elimination $\delta(R)$

when it cannot be performed in the main memory

Sort-based algorithm: essentially TPMMS where duplicates are eliminated before output

Phase 1: Sort parts of size *MMBuffers* individually with all attributes as the sort key

Phase 2: Merge the parts eliminating duplicate tuples when output

Cost: $3 * \text{Blocks}(R)$

Requirement: $\sqrt{\text{Blocks}(R)} \leq \text{MMBuffers}$

Two-pass hash-based duplicate elimination

Duplicate elimination $\delta(R)$

when it cannot be performed in the main memory

Hash-based algorithm: exploit that duplicates will have the same hash value

Partition the relation into buckets, write buckets back to disk
Run one-pass duplicate elimination individually on each bucket

Cost: $3 * Blocks(R)$

Requirement: The size of each bucket ($MMBuffers - 1$ in total) can't be more than $MMBuffers$:

$$Blocks(R) \leq MMBuffers * (MMBuffers - 1)$$

$$\text{so essentially } \sqrt{Blocks(R)} \leq MMBuffers$$

(Assuming equal distribution over buckets, i.e., good hash function)

The same bound as sort-based, but quite different approach

Grouping with Aggregation

Grouping and Aggregation γ_L is very similar to duplicate elimination δ

One-pass: same idea, keep a tuple for each group in memory

Cost: $Blocks(R)$

Requirement: $Blocks(\gamma_L(R)) \leq MMBuffers - 1$ (little more for AVG)

Sort-based two-pass: same idea, except that the sort key is the grouping attributes

Cost : $3 * Blocks(R)$

Requirement: $\sqrt{Blocks(R)} \leq MMBuffers$ (approximately)

Hash-based two-pass: same idea, except that the hashed key is the grouping attributes

Cost : $3 * Blocks(R)$

Requirement: $\sqrt{Blocks(R)} \leq MMBuffers$

(assuming good hash function)

3.2 Binary operations

Bag union $R \cup^b S$ is essentially tuple-at-a-time

1. Retrieve and output all blocks of R one by one
2. Retrieve and output all blocks of S one by one

Cost: $Blocks(R) + Blocks(S)$

Requirement: $MMBuffers \geq 1$

One-pass set union

Set union $R \cup S$: need to eliminate duplicates

Keep the smaller of R and S in $MMBuffers - 1$ buffers, use the other buffer to pass through the bigger one

Assuming $Blocks(R) \leq Blocks(S)$,

load all blocks of $Blocks(R)$ into the $MMBuffers - 1$ buffers,
output all tuples in $Blocks(R)$ blocks (keeping in buffers)

Repeat:

1. Retrieve the next block of S into the one buffer
2. For each tuple t of the S buffer,
search for t in the R buffers,
output t if not found

Until all blocks of S are processed

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Two-pass sort-based set union

(Set) union $R \cup S$

when it cannot be performed in the main memory

Sort-based algorithm: essentially duplicate elimination by TPMMS for $R \cup S$

Phase 1: Sort parts of $R \cup S$ of size $MMBuffers$ individually with all attributes as the sort key

Phase 2: Merge the parts eliminating duplicate tuples when output

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{Blocks(R) + Blocks(S)} \leq MMBuffers$

Two-pass hash-based set union

(Set) union $R \cup S$

when it cannot be performed in the main memory

Hash-based algorithm: using that duplicates have same hash value

Partition R and S into buckets separately, write buckets to disk

Run one-pass set union for each pair of buckets with same value

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: at most $MMBuffers - 1$ buckets (for both R and S),
 $\min(Blocks(R_i), Blocks(S_i)) \leq MMBuffers - 1$ for each bucket i :

essentially $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$

(Assuming equal distribution over buckets,

and that one of R , S is smaller than the other in all buckets)

Better than sort-based, but with an essential assumption

Intersection and difference (set and bag)

Same ideas as for set union:

One-pass:

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Two-pass sort-based:

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{Blocks(R) + Blocks(S)} \leq MMBuffers$

Two-pass hash-based:

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$ (almost)

(under the assumptions; note also that the bounds are symmetric even for non-commutative difference)

One-pass natural join

Natural join $R \bowtie S$

when the smaller relation fits to the main memory

Keep the smaller of R and S in $MMBuffers - 1$ buffers, use the other buffer to pass through the bigger one:

Assuming $Blocks(R) \leq Blocks(S)$,

load all blocks of $Blocks(R)$ into the $MMBuffers - 1$ buffers

Repeat:

1. Retrieve the next block of S into the one buffer
2. For each tuple t in the S buffer,
search for the tuples joining with t in the R buffers,
output all join results

Until all blocks of S are processed

Cost: $Blocks(R) + Blocks(S)$

Requirement: $\min(Blocks(R), Blocks(S)) \leq MMBuffers - 1$

Nested-loop natural join

Natural join $R \bowtie S$

generalisation of one-pass

Apply one-pass join to the parts of the smaller of R and S separately, using $MMBuffers - 1$ buffers:

Assuming $Blocks(R) \leq Blocks(S)$,

Repeat:

1. load the next $MMBuffers - 1$ blocks of $Blocks(R)$
into the $MMBuffers - 1$ buffers
2. apply one-pass join to these buffers and full S

Until all blocks of R are processed

Cost: $Blocks(R) + Blocks(S) * \lceil Blocks(R) / (MMBuffers - 1) \rceil$

Requirement: $2 \leq MMBuffers$

(essentially, no space requirement, but quadratic)

Two-pass sort-based natural join

Natural join $R \bowtie S$ via two-pass sorting

First sort separately, then join:

Sort R and S separately on the join attributes by TPMMS

Assign $MMBuffers/2$ buffers to R , same for S

Repeat:

1. fill empty R buffers with next R blocks, same for S
2. output all joined tuples with the smallest sort values,
delete contributing tuples

Until all blocks of R and S are processed

Cost: $5 * (Blocks(R) + Blocks(S))$ (4 passes for TPMMS, 1 for join)

Requirement 1: $\sqrt{\max(Blocks(R), Blocks(S))} \leq MBuffers$

Requirement 2: at most $MMBuffers/2$ blocks with the same join value in each of R and S

Two-pass hash-based natural join

Natural join $R \bowtie S$ via hashing

Hash-based algorithm: using that duplicates have same hash value

Partition R and S into buckets separately hashing the join values

Run one-pass join for each pair of buckets with same value

Cost: $3 * (Blocks(R) + Blocks(S))$

Requirement: at most $MMBuffers - 1$ buckets (for both R and S),
 $\min(Blocks(R_i), Blocks(S_i)) \leq MMBuffers - 1$ for each bucket i :

essentially $\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$

(Assuming equal distribution over buckets,
and that one of R, S is smaller than the other in all buckets)

Two-pass hybrid hash-based natural join

Natural join $R \bowtie S$ via hashing with improvement

Improvement: Keep one bucket of smaller relation in main memory, and join on the fly when hashing the larger relation (assuming that the number of buckets allow this)

Cost: $(3 - 2MMBuffers/Blocks(R)) * (Blocks(R) + Blocks(S))$

(**Cost of hash-based:** $3 * (Blocks(R) + Blocks(S))$)

Homework: understand this advantage

Requirement: essentially

$$\sqrt{\min(Blocks(R), Blocks(S))} \leq MMBuffers$$

(with same assumptions plus

number of buckets + size of the R bucket $\leq MMBuffers$)

One index based natural join

Natural join $R \bowtie S$ using an index

Iterate over R , get relevant tuples of S using index on join attributes A :

Repeat:

1. read next blocks of R into $MMBuffers - 1$ buffers
2. for each tuple t in the buffers,
 retrieve all tuples of S joining with t using index and last block
 output joined tuples, remove t from the buffers

Until all blocks of R are processed

Cost (estimate) for clustered index (i.e., S is sorted on A):

$$Blocks(R) + Tups(R) * \lceil Blocks(S) / Vals(S.A) \rceil$$

Cost (estimate) for non-clustered index:

$$Blocks(R) + Tups(R) * \lceil Tups(S) / Vals(S.A) \rceil$$

Requirement: $2 \leq MMBuffers$

Useful only if R is small and $Vals(S.A)$ is large

(e.g., A is a primary key for S)

Zig-zag (two index based) natural join

Natural join $R \bowtie S$ using two indexes

Join the indexes on the join attributes A for both R and S :

1. identify relevant blocks of R and S using the indexes
2. load the blocks with the same join values
and construct the output

Cost: at most $Blocks(R) + Blocks(S)$ (as one pass)

Requirement: blocks for each join value (for R and S together)
should be at most $MMBuffers$

(an estimation formula can be written
via $Vals(R.A)$ and $Vals(S.A)$
relying on the uniformity assumption)

Natural join algorithms: Comparison example

Natural join $R \bowtie S$ on a common attribute A ,
both with clustered indexes

Statistics: $Tups(R) = 10,000$, $Blocks(R) = 1,250$, $Vals(R.A) = 100$
 $Tups(S) = 5,000$, $Blocks(S) = 625$, $Vals(S.A) = 10$

	Cost (often estimation)	Requirement for <i>MMBuffers</i>
One-pass	1875	626
Nested-loop	9375	2
Two-pass sort-based	9375	36
Two-pass hash-based	5625	25
Hybrid hash-based	5019	25
One index-based on S	625,000	2
One index-based on R	62,500	2
Zig-zag	1875	76

Note: cost of nested-loop and hybrid hash-join depends on *MMBuffers*;
we take $MMBuffers = 101$

4. Summary

Algorithms summary

One-pass: good, but applies generally when one argument **fits into the memory**

Two-pass: much **lighter** memory requirements

- Sort-based:
give **sorted results**, which can be utilised later
- Hash-based:
usually requires **less memory** (only the smaller counts)
relies on good **hash function** (equal distribution over buckets)
- Index-based:
requires **indexes**
great for **selection**
often efficient for **join**

N-pass: generalisation of **two-pass** for even bigger relations

N -pass algorithms

N -pass: generalisation of **two-pass** for even bigger relations

Example: $Blocks(R) = 1,000,000$

- TPMMS requires $\sqrt{Blocks(R)} \leq MMBuffers$ so $MMBuffers \geq 1,000$
- for bigger R , two passes are not enough

N -PMMS sorting:

- recursive TPMMS
- **Cost:** $(2N - 1) * Blocks(R)$
- **Requirement:** $\sqrt[N]{Blocks(R)} \leq MMBuffers$

Similar ideas applies to **hash-based** algorithms

Back to query plans

Logical query plan: RA expression

Physical query plan: Logical query plan where an **algorithm** is assigned to each operation

For simplicity, we assumed that we have a **logical plan** constructed, and studies algorithms for RA operations **'in isolation'** (often several algorithms for one operation with different costs and requirements)

In reality, these costs are used to **choose logical plans**
(even more search space)

We may have problems with **evaluation**:

our algorithms rely on **assumptions**, which may not hold in reality,
and we may have to use another algorithm

(or even adapt the already running one losing the performance)

5. Query plans in real DBMSs

Real DBMSs **estimate costs** based on their knowledge on the system
(block size, main-memory operation costs, etc.)

For example, in PostgreSQL the defaults are something like

```
seq_page_cost = 1
random_page_cost = 4
cpu_tuple_cost = 0.01
cpu_operator_cost = 0.0025
cpu_index_tuple_cost = 0.005
```

Usually updated **automatically**,
but sometimes can be **manually** changed (for estimation)

PostgreSQL query plans analysis

PostgreSQL has `EXPLAIN` command that provides information on query plans and their costs (**Play with it**, see accessible manual at https://wiki.postgresql.org/wiki/Using_EXPLAIN)

Example: `EXPLAIN SELECT * FROM Post ORDER BY Body LIMIT 50;`

```
Limit (cost = 23283.24..23283.37 rows = 50 width = 422)
-> Sort (cost = 23283.24..23859.27 rows = 230412 width = 422)
Sort Key: body
-> Seq Scan on post (cost = 0.00..15629.12 rows = 230412 width = 422)
```

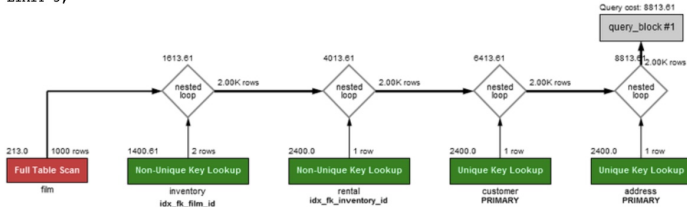
Has parameters:

- `VERBOSE`: more information
- `ANALYZE`: also runs the query and compares plan with reality
- ...

Quickly become unreadable, but there are **visualisation** tools
(e.g., <http://tatiyants.com/pev/>)

MySQL query plan analysis

```
SELECT CONCAT(customer.last name, ' ', customer.first name) AS customer, address.phone, film.title
FROM rental INNER JOIN customer ON rental.customer id = customer.customer id
INNER JOIN address ON customer.address id = address.address id
INNER JOIN inventory ON rental.inventory id = inventory.inventory id
INNER JOIN film ON inventory.film id = film.film id
WHERE rental.return date IS NULL
AND rental date + INTERVAL film.rental duration DAY < CURRENT_DATE()
LIMIT 5;
```



Much **less** information

What have we learned in Part 1?

```
SELECT DISTINCT title FROM StarsIn WHERE starName IN  
  (SELECT name FROM MovieStar WHERE birthDate LIKE '%1960');
```

⌋ parsing

(parse tree)

⌋ compilation

$$\pi_{title}(\sigma_{starName=name}(\mathbf{StarsIn} \times \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar}))))$$

⌋ optimisation

$$\pi_{title}(\mathbf{StarsIn} \bowtie_{starName=name} \pi_{name}(\sigma_{birthDate \text{ LIKE } '%1960'}(\mathbf{MovieStar})))$$

⌋ evaluation over tables

(Result)

IN3020&4020 – Database Systems (2024)

Part 2: Transaction management

Lectures 16–18: Transaction processing concepts

11, 12, 18 March

Egor V. Kostylev and **Martin Giese**

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases

- Transaction processing concepts (3 lectures)
- Concurrency control techniques (3 lectures)
- Database recovery techniques (2 lectures)

Part 3. NoSQL DBMS

Part 2. **Transaction management in relational databases**

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): Part 9 (close to my presentation)

- **Transaction processing concepts** (3 lectures)

Chapter 20

- **Concurrency control techniques** (3 lectures)

Chapter 21

- **Database recovery techniques** (2 lectures)

Chapter 22

Database Systems: the Complete Book by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Sections 6.6, 19

etc.

1. Motivation and introduction
2. Desirable properties of transactions (ACID)
3. Transactions, schedules, and conflicts
4. Characterising schedules based on serialisability
5. Characterising schedules based on recoverability
6. Isolation levels

1. Motivation and introduction

- Transactions (informally):
mechanism for managing **logical units** of data processing:
independent on others, all or nothing
- **Examples:**
 - a single data retrieval query
 - a sequence of data manipulation queries that should be executed together
- Transaction management (or processing) systems:
systems with **large databases** and many **concurrent users**
require **high availability** and **fast response time**
- **Examples:**
 - banking, airline booking, online retail, stocks

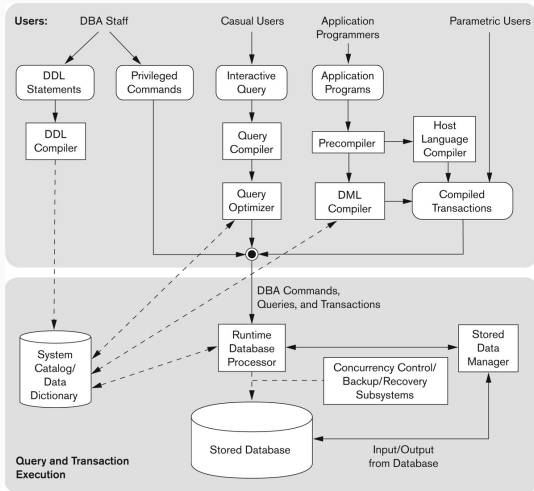
Single-user vs multi-user, challenges

- Single-user DBMS:
 - at most **one user** at a time
 - (usually, personal computer)
- Multi-user DBMS:
 - many users** (processes with own computation)
 - with **concurrent access** to the same data
 - (usually, servers with many CPUs,
 - but may be handled by one CPU
 - with interleaving concurrency)

Both need **transaction management**, with two **challenges**:

- **concurrency control** (transactions independent of each other)
- **fail recovery** (each transaction executed all or nothing)

The place of transaction management



Transaction idea

Transaction (informally):

- program that forms a **logical unit** of database processing
- has its own **memory** and **computation ability**
- includes one or more **access operations** to (shared) database
(e.g., retrieval, insertion, deletion)
- for us, usually a **sequence** of operations
(no branching, no cycles, etc.)
- **boundaries** can be specified by **begin** and **end statements**
(if something goes wrong on the way, effects roll back)
- an application program may have **several** transactions
(even parallel)
- may be **read-only** (only data retrieval) and **read-write**

Data items and granularity

In [Part 2](#), we use a simple data model:

a database is a set of [named data items](#)

- can be a [tuple](#) (record), [block](#), [attribute value](#)
- can be uniquely [identified](#) by name (address, tuple ID, etc.)
- can be [read](#) and [written](#) by name

Item [granularity](#):

- the [size](#) of a data item

[Transaction processing](#) concepts are
[independent](#) of item granularity

Main transaction operations

read(X)

- **reads** an item named X from the **global** database into a **local** program variable named X
- includes finding the address of the block on the disk (or in cache) with X , and copying to a main memory buffer

write(X)

- **writes** the value of **local** program variable named X into the **global** database item named X
- includes finding the address of the block on the disk (or in cache) with X , read it to the local memory buffer, modify it, and write it back (to the disk or cache)

Program local operations (for example, update $X := X + 50$)

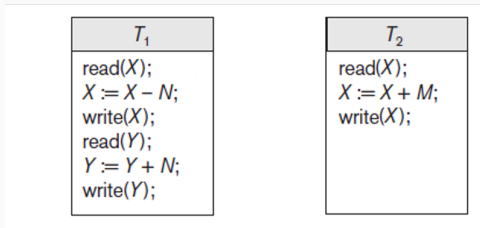
We will see **other** (transaction management) operations later

Example transactions

Let X and Y be the numbers of **reserved seats** in two flights
(stored in a database)

Transaction T_1 **transfers** N reservations from X to Y

Transaction T_2 **reserves** M seats in X



Important:

- updates are local, the (only) database is not updated until the new value is written
- some commands are often omitted if they are not relevant (both local and transaction management)

Why concurrency control is needed?

Potential **concurrency control** problems (or phenomena):

1. Dirty write (or lost update)
2. Dirty read (or temporary update)
3. Non-repeatable read
4. Incorrect summary (or phantom phenomena)
5. ...

(**Other** variations of these problems can be found in the literature)

Example transactions

Let X and Y be the numbers of **reserved seats** in two flights

Transaction T_1 **transfers** N reservations from X to Y

Transaction T_2 **reserves** M seats in X

T_1
<pre>read(X); X := X - N; write(X); read(Y); Y := Y + N; write(Y);</pre>

T_2
<pre>read(X); X := X + M; write(X);</pre>

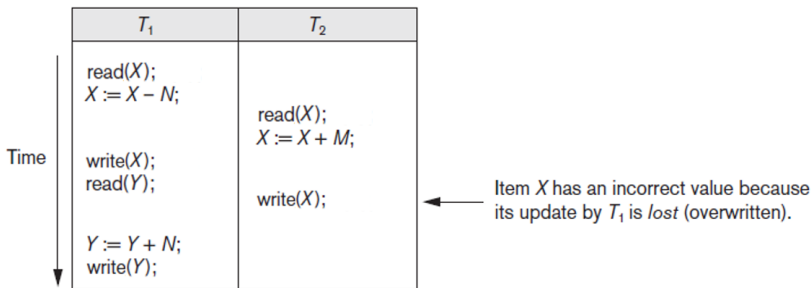
What can go wrong when interleave these transactions?

Dirty write (lost update)

Two transactions update (read and write) the **same** item,
second update starts before the first is complete

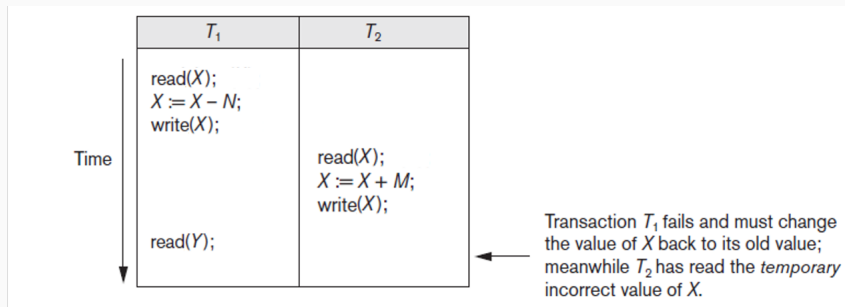
(updates are **interleaved**)

Result: **incorrect value** (update is lost)



Dirty read (temporary update)

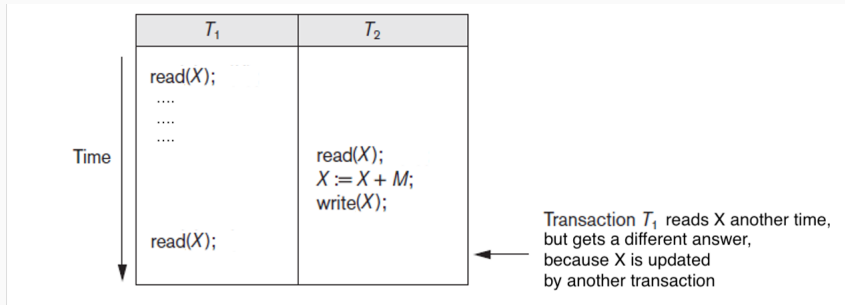
A transaction **updates** an item,
new value is **used** by another transaction,
the first transaction **fails** (see below) and its update is **rolled back**
Result: the second transaction **relies** on an incorrect value



Non-repeatable read

One transaction reads the same item **twice**,
another transaction **changes** its value in between

Result: the first transaction gets **different** values of the **same** item



Incorrect summary (Phantom phenomena)

One transaction calculates an **aggregate summary**, while other transactions **update** some involved items

Result: the aggregate is **inconsistent**

T_1	T_3
<pre>read(X); X := X - N; write(X); read(Y); Y := Y + N; write(Y);</pre>	<pre>sum := 0; read(A); sum := sum + A; ⋮ read(X); sum := sum + X; read(Y); sum := sum + Y;</pre>

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why recovery is needed?

Transaction idea (reminder):

- program that forms a **logical unit** of database processing
- includes one or more database **access operations**
(e.g., retrieval, insertion, deletion)
- **boundaries** can be specified by **begin** and **end statements**
(if something goes wrong, effects roll back)

Should be either

- **Committed:**
evaluated in full and all effects permanently recorded
- **Aborted:**
no effect on the database (everything done rolled back)

Should take into account and use **cache** and **log**
(which we will introduce later)

Why recovery is needed?

Types of failures:

- Computer failure (system crash):
hardware, software, network errors
- Transaction failure:
division by zero, integrity constraint violation, user interrupt, etc.
- Local transaction errors:
no data found, programmed exception, etc.
- Concurrency control enforcement:
serialisability violation, deadlock resolving, etc.
- Disk failure:
errors with disk reads or writes
- Physical problems:
power cut, fire, catastrophe, etc.

To develop a principled machinery (formalise concepts, develop algorithms, etc.) for **transaction management** system:

- **concurrency control** subsystem
- **fail recovery** subsystem

2. Desirable properties: ACID principles

ACID:

- Atomicity

Transaction performed in its entirety or not at all

- Consistency

The database should always remain consistent

- Isolation

Transaction should not interfere with other transactions

- Durability

Changes of committed transactions must persist

Atomicity: Transaction performed in its entirety or not at all

- ensured by transaction recovery subsystem of a DBMS
- the recovery technique must undo any effects of a failed (aborted) transaction
- done by using the log and writing back the old values (we will discuss the log and techniques)
- cooperation with concurrency control subsystem is also necessary

Consistency: The database should **always** remain **consistent**

- each transaction should bring database from one **consistent state** to another **consistent state**
- **consistent state** (reminder): the database satisfies the **integrity constraints** specified in the schema as well as any other **constraints** on the database that should hold
- the responsibility of the database **program**
(i.e., the programmer)
- transactions can use **abort** instructions, leading to **rollbacks**
(handled by **recovery subsystem**)

Isolation: Transaction should not **interfere** with other transactions

- ensured by the **concurrency control** subsystem
- ideally, we should avoid **all problems** (**dirty writes and reads, non-repeatable reads, phantoms, etc.**)
- this may be too **expensive** (in terms of efficiency), so often may be relaxed in one way or another (**isolation levels**, we will see)
- one of the most **difficult** and **interesting** issues for transaction management

Durability (or permanency): Changes of committed transactions must **persist**

- ensured by the **recovery** subsystem
- after a **crash**, the **log** is used to **restore** committed transactions and **roll back** uncommitted ones
- **cache** complicates restoration

3. Transactions and schedules

Transactions

Transaction is a sequence of **operations**, an atomic unit of work

Structure of a **committed** (successful) transaction:

1. **begin** ← marks the beginning of transaction execution
2. one or several **read(X)**, **write(X)**, local operations (e.g., var updates), transaction control operations (e.g., locks), etc.
3. **end** ← mark the end of transaction execution
4. one or several operations checking consistency, serialisability, etc.
5. **commit** ← successful completion, changes cannot be undone

Structure of an **aborted** (unsuccessful) transaction:

same beginning, but ends at any point with

- N. **abort** ← unsuccessful completion, all changes must be undone

A **partial** transaction:

a beginning of one above ('waiting' for next operation)

Transaction notation

Transaction is a sequence of **operations** (not arbitrary)

- **begin**, **end**, **commit**,
abort, **read(X)**, **write(X)**, variable updates, etc.
- can be **committed**, **aborted**, or **partial**
(depending on the last operation)

Often we

- use **abbreviations**:
b, **e**, **c**, **a**, **r(X)**, **w(X)** (var updates the same)
- **omit** operations **irrelevant** in the context
(in most cases, only **r(X)** and **w(X)** are relevant)
- when considering **several** transactions,
append transaction **ids** to operations:
b₁, **r₁(X)**, **r₂(X)**, **w₃(X)**

Example transactions

Let X and Y be the numbers of reserved seats in two flights

T_1	T_2
<pre>read(X); X := X - N; write(X); read(Y); Y := Y + N; write(Y);</pre>	<pre>read(X); X := X + M; write(X);</pre>

(Some operations are already omitted here)

May be written as

$T_1: r_1(X), w_1(X), r_1(Y), w_1(Y)$

$T_2: r_2(X), w_2(X)$

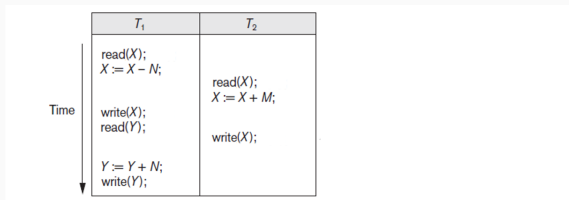
Schedules

Schedule (or history, execution plan) S of transactions T_1, \dots, T_n :
a (total) **ordering** of the operations of T_1, \dots, T_n

- operations of different transactions can interleave
- operations of each T_i are in the same order as in T_i

Complete schedule: all T_1, \dots, T_n are **committed** or **aborted**

Example: S_a : $r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y)$



Example (complete): S_b : $r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), a_1, c_2$

4. Characterising schedules based on serialisability

For **isolation** (transaction should not interfere with other transactions), we need to be able to ensure that all transactions of a schedule are executed **correctly**

We will **characterise** the types of schedules
that are considered to be correct:

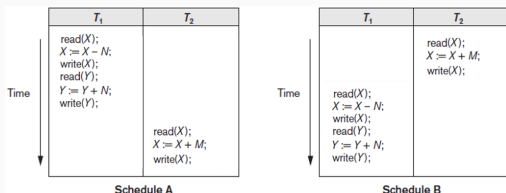
serial, (conflict-)serialisable, view-serialisable

Serial schedules

Serial schedule intuitively: no interleaving

Serial schedule: no transaction begins when there is an active (non-committed and non-aborted) transaction

(i.e., all the operations of every transaction are executed consecutively, and only one transition at a time is active)



A: $r_1(X), w_1(X), r_1(Y), w_1(Y), c_1, r_2(X), w_2(X), c_2$

B: $r_2(X), w_2(X), c_2, r_1(X), w_1(X), r_1(Y), w_1(Y), c_1$

Every serial is considered correct

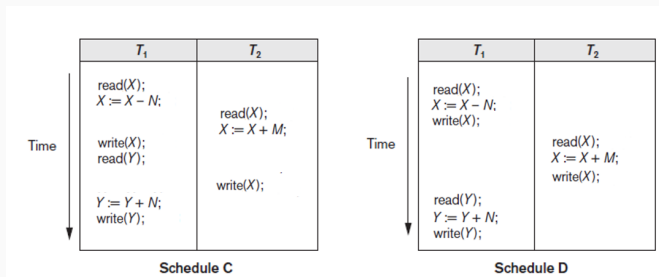
Limit (essentially no) concurrency: unacceptable

Serialisable schedules

Serialisable schedule: **equivalent** to a serial schedule

We need to define transaction **equivalence**

Examples:



C should not be serialisable, D should be

Most common is **conflict equivalence** (in two slides),
but there are **other** possibilities

Naive approaches to equivalence

Naive approaches to equivalence:

- result equivalence:
two schedules give the same result on the current database
- semantic equivalence:
two schedules give the same result on all possible databases

Question: What are the problems with these approaches?

We will develop syntactic notions of equivalence, which guarantee semantic (and hence result) equivalence. The first and the main is based on the notion of a **conflict**.

Intuitively: a **conflict** is a pair of operations in a schedule such that

- they are from **different transactions**
- **flipping** them can result in a **different outcome**

Formally: two operations in a schedule is

- a **read-write conflict** if they are $r_i(X)$ and $w_j(X)$ with $i \neq j$ (in any order)
- a **write-write conflict** if they are $w_i(X)$ and $w_j(X)$ with $i \neq j$

Examples:

- $r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y)$
- $r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), a_1$
- $r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), a_1$

(You may sometimes see **intra-transaction conflicts** in literature,
but we do not need them in our formalisation)

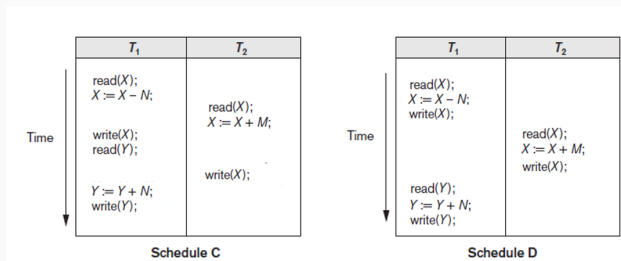
Misleading name: **Nothing wrong per se!**

Conflict equivalence

Two schedules are **conflict equivalent** if

- they are schedules of the **same transactions**
- if the **relative order** of every two conflicts (read-write, write-write) is the **same** in both schedules

(Conflict-)serialisability: serialisability based on **conflict equivalence**



Schedule **C** is **not (conflict-)serialisable**

Schedule **D** is **serialisable** (equivalent to $T_1; T_2$)

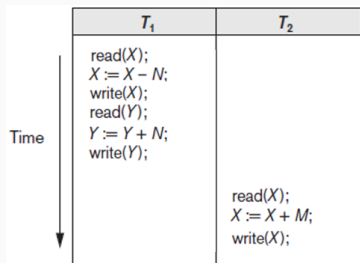
Testing for serialisability

Simple **algorithm** for testing serialisability of a schedule S

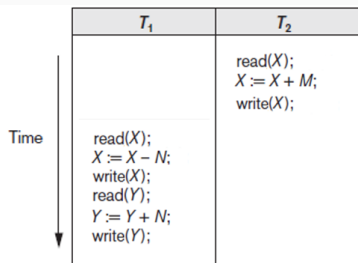
1. Construct **precedence graph** of a S :
 - **nodes**: **transactions** in S
 - (directed) **edges**: **conflict pairs** of operations
(from the earlier to the later)
(i.e., operations from different transactions
one of which is write)
2. Return **serialisable** if and only if
the precedence graph has no **cycles**

Sanity check examples

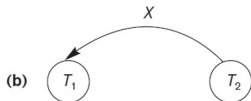
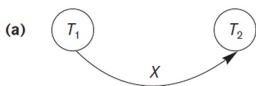
Serial schedules (obviously) have **acyclic** precedence graph



Schedule A



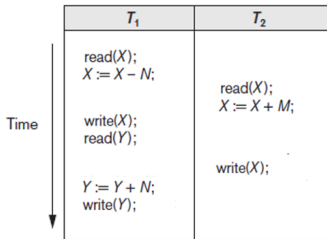
Schedule B



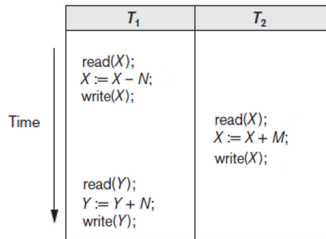
(Labels X are not essential, just for reference)

More examples – 1

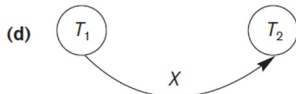
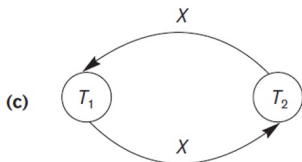
Non-serialisable and serialisable schedules:



Schedule C

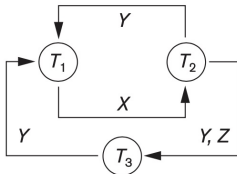


Schedule D



More examples – 2

	Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read(X); write(X);	read(Z); read(Y); write(Y);	read(Y); read(Z);
	read(Y); write(Y);	read(X); write(X);	write(Y); write(Z);



Equivalent serial schedules

None

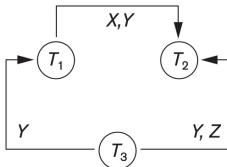
Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

Cycle $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

More examples – 3

	Transaction T_1	Transaction T_2	Transaction T_3
Time ↓	read(X); write(X);		read(Y); read(Z);
	read(Y); write(Y);	read(Z);	write(Y); write(Z);
		read(Y); write(Y); read(X); write(X);	



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Serialisability based on conflict-equivalence is rather **strong**,
and can be **relaxed**

Example: $r_1(X), w_2(X), w_1(X), w_3(X), c_1, c_2, c_3$

Not conflict-serialisable, but the effect is equivalent to serial
 $r_1(X), w_1(X), c_1, w_2(X), c_2, w_3(X), c_3$
(due to **blind write** $w_3(X)$)

View equivalence

Schedules S and S' are **view equivalent** if

- they are schedules of the **same transactions**
- each $r(X)$ reads from **the same place** in S and S'
(i.e., in both schedules either X has been previously written by the same $w(X)$ or not written at all)
- the **last write** of each X is also the same

View-serialisability: serialisability based on **view equivalence**

Example: $r_1(X), w_2(X), w_1(X), w_3(X), c_1, c_2, c_3$

Each **conflict-serialisable** is **view-serialisable**

Each **view-serialisable** is **semantic-serialisable** (so **result-serialisable**)

Difficult to check view-equivalence (**NP-complete**)

By default, we rely on conflict-equivalence,

and 'serialisability' below means 'conflict-serialisability'

Serialisability in concurrency control

Every **serial** schedule is **serialisable**, but not other way round

Serialisable schedules give benefit of **concurrent execution** without giving up any **correctness**

Difficult to **test** for serialisability in practice (even conflict-serialisability)

- system load, time of transaction submission, and process priority affect ordering of operations
- often we need to ensure serialisability **before** the transactions complete

DBMSs enforce concurrency control **protocols** that ensure serialisability

5. Characterising schedules based on recoverability

Recoverability informally

For **durability** (changes of committed transactions must persist), we need to be able to **recover** schedules from transaction and system failures

We will see techniques for this (in a couple of weeks), but before this it is important to **classify** schedules with respect to recoverability:

- **impossible** to recover
(we will see examples)
- **possible** to recover
recoverable (we will define formally and see examples)
- **easy** to recover
cascadeless, strict (we will define and see examples)

Recoverable schedules

Transaction T reads from (different) transaction T' in a schedule if an item X is first written by T' and then read by T (and no transactions write X in between)

Schedule S is recoverable if no transaction T in S commits until all transactions T' such that T reads from T' have committed

Examples:

$r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), c_2, w_1(Y), c_1$
recoverable

$r_1(X), w_1(X), r_2(X), r_1(Y), w_2(X), w_1(Y), c_2, a_1$
non-recoverable

$r_1(X), w_1(X), r_2(X), r_1(Y), w_2(X), w_1(Y), a_1, c_2$
still non-recoverable

$r_1(X), w_1(X), r_2(X), r_1(Y), w_2(X), w_1(Y), a_1, a_2$
recoverable

Cascadeless schedules

Recoverable schedules may need **cascading aborts** (**cascading rollbacks**) to recover—that is, an (non-committed) transaction has to abort since it has read from another failed transaction

Example: $r_1(X), w_1(X), r_2(X), r_1(Y), w_2(X), w_1(Y), a_1, a_2$

This may affect many transactions, and rollbacks may be expensive

Cascadeless schedule idea: no cascade rollbacks

Cascadeless schedule: every transaction in the schedule reads only from committed transactions

(i.e., for each $r(X)$, X can be previously written by the same transaction, written by another **committed** transaction or have not been written before)

Example:

$r_1(X), w_1(X), r_1(Y), r_2(Z), w_1(Y), c_1, w_2(Z), r_2(X), w_2(X), c_2$

Strict schedules

Cascadeless schedules may still have fairly complicated recovery protocols, because we cannot just restore the values of all writes of an aborted transaction

Example: $w_1(X), w_2(X), a_1$

This may all affect many transactions (no cascades though)

Strict schedule: no transaction can read or write an item X until the previous write of X is committed or aborted

Examples:

$w_1(X), c_1, w_2(X)$

$w_1(X), a_1, w_2(X)$

Every **strict** schedule is **cascadeless**

Every **cascadeless** is **recoverable**

We will study **recovery techniques** for different types of schedules
in a couple of lectures

6. Isolation levels

Back to problems (phenomena)

Potential **concurrency control** problems (or phenomena):

1. **Dirty write** (or **lost update**): ... $r_1(X), \dots, w_2(X), \dots, w_1(X)$...
not **conflict-serialisable**
2. **Dirty read** (or **temporary update**): ... $w_1(X), \dots, r_2(X), \dots, a_1$...
not **recoverable**
3. **Non-repeatable read**: ... $r_1(X), \dots, w_2(X), \dots, r_1(X)$, ...
not **conflict-serialisable**
4. **Phantoms** (incl. **incorrect summary**): ... $r_1(X) \dots w_2(Y) \dots r_1(Y) \dots$
where X and Y contribute to query of T_1 in the same way
need to treat this by **other mechanisms**

Isolation levels

It is not always necessary to ensure no problems of all types

- in some applications we can tolerate some problems
- for the sake of more concurrency (i.e., efficiency)

Simple isolation level hierarchy of schedules

(and also transaction management protocols):

Level 0: no dirty reads

Level 1: no dirty writes

Level 2: no dirty reads, no dirty writes

Level 3 (true isolation): no dirty reads, no dirty writes,
no non-repeatable reads

A protocol that ensure

recoverable and serialisable schedules is Level 3

Isolation levels in SQL

SQL standard concentrate on

dirty reads, non-repeatable reads, and phantoms

Isolation levels in SQL ('yes' is allowed):

Isolation Level	Type of Violation		
	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Note: SERIALIZABLE here is different from **conflict-serialisable**

(Real DBMSs do **not** always follow the standard:

for example, no **dirty writes** are usually also ensured)

SQL transaction language

SQL has a [language](#) for transactions

(systems may not follow the standard in all details)

Example:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE                               ← access mode
    DIAGNOSTIC SIZE 5                         ← diagnostic area size
    ISOLATION LEVEL SERIALIZABLE;           ← isolation level
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Access mode: READ ONLY (only SELECT queries) or READ WRITE

Diagnostic size: number of statements with diagnostic messages

Note: need to handle schedules with transaction of different levels

What have we learned?

Basics of transaction management:

- **ACID principles** (Atomicity, Concurrency, Isolation, Durability)
- **main concepts** (transactions, schedules, conflicts)
- **recoverable schedules** (including cascadeless and strict)
- **serialisable schedules** (including conflict- and view-serialisable)
- **isolation levels** (several types)

In the rest of Part 2:

1. techniques for **concurrency control**
2. techniques for **recovery**

IN3020&4020 – Database Systems (2024)

Part 2: Transaction management

Lectures 19–21: Concurrency control techniques

19 March, 2, 8 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases

- Transaction processing concepts (3 lectures) ✓
- **Concurrency control techniques** (3 lectures)
- Database recovery techniques (2 lectures)

Part 3. NoSQL DBMS

Part 2. **Transaction management in relational databases**

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): **Part 9** (close to my presentation)

- Transaction processing concepts (3 lectures)
Chapter 20
- **Concurrency control techniques** (3 lectures)
Chapter 21
- Database recovery techniques (2 lectures)
Chapter 22

Database Systems: the Complete Book by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): **Parts of Section 18**

etc.

Reminders – 1

Transaction is a sequence of **operations** (not arbitrary)

- **begin**, **end**, **commit**,
abort, **read**(X), **write**(X), variable updates, etc.
- we use **abbreviations** (**b**, **e**, **c**, **a**, **r**(X), **w**(X))
- we may **omit** operations (e.g., use only **r**(X) and **w**(X))
- we often append transaction **ids** (e.g., **b**₁, **r**₁(X), **r**₂(X), **w**₃(X))
- transactions read from and write to the common disk,
but have their own main memory

Can be **committed**, **aborted** or **partial**

Schedule (history, or execution plan) S of transactions T_1, \dots, T_n is a (total) **ordering** of the operations of T_1, \dots, T_n

- operations of different transitions can interleave
- operations of each T_i are in the same order as in T_i

Complete schedule: all T_1, \dots, T_n are **committed** or **aborted**

ACID principles:

- **Atomicity**: Transaction performed in its entirety or not at all
- **Consistency**: The database should always remain consistent
- **Isolation**: Transaction should not interfere with others
- **Durability**: Changes of committed transactions must persist

We agreed to ensure isolation by enforcing **(conflict-)serialisability**:

- schedule is **serial** if **no** transaction **begins** when there is another **active** (non-committed and non-aborted) transaction
- schedule is **(conflict-)serialisable** if it is **(conflict-)equivalent** to a serial one
- two schedules (of the same transactions) are **conflict-equivalent** if the **relative order** of every two conflicts is the **same** in both
- a conflict is a pair of operations from different transactions at least one of which is **write**

Ensuring serialisability in DBMSs

Naive approach:

- run all transactions in an **arbitrary way**
- construct the **precedence graph** in the process
- when done, check if the graph is **acyclic**
- **abort** and restart everything if not

Obvious drawbacks

Instead, DBMSs arrange transactions
(individually and in the schedule)
using **concurrency control protocols** (i.e., set of rules)
that ensure serialisation by **construction**

In this **set of lectures**, we consider such **protocols**

1. Serialisability via two-phase locking
 - 1.1. Types of locks
 - 1.2. Two-phase locking protocol
 - 1.3. Deadlocks
2. Serialisability via timestamps
3. Multiversioning for concurrency control
4. Concurrency with multiple granularity
5. Other issues and conclusion

1. Two-phase locking

1.1. Types of locks

A lock: a **variable** associated with a **data item**

- **unique** $LKD(X)$ for each data item X
- describes **status** for operations that can be applied

We will consider

- **binary locks:**
 $LKD(X)$ takes values **locked** or **unlocked** (or 1 and 0)
- **shared/exclusive locks:**
 $LKD(X)$ takes values **read-locked**, **write-locked**, or **unlocked**

These variables are managed by
concurrency control subsystem (transaction manager)

Binary locks:

for each item X , variable $LKD(X)$ can have one of the **two values**:

- $LKD(X)$ is **locked** (or 1) means that X is **locked** by a transaction and cannot be accessed by any other transaction
- $LKD(X)$ is **unlocked** (or 0) means that X is **available** and can be accessed when requested
- schedules **start** with all **unlocked**

Binary locks in transactions

Transaction with binary locks: can use two operations `lock(X)` and `unlock(X)` following **several rules**:

- all `read(X)` and `write(X)` must be between the two
- is well-formed: no unlocking without locking, etc.

These operations are **implemented** such that

- when `lock(X)` is issued
 - if $LKD(X) = 1$, then the transaction is forced to wait for $LKD(X) = 0$ (in the schedule)
 - if $LKD(X) = 0$, it is set to 1 and the transaction can proceed
- when `unlock(X)` is issued, $LKD(X)$ is set to 0

run by transaction manager (concurrency subsystem), see below

As a result, certain schedules are **disallowed**

Example: Transactions with binary locks

<u>T_1</u>	<u>T_2</u>
lock(X);	lock(X);
read(X);	lock(Y);
$X := X - 1$;	read(X);
write(X);	unlock(X);
unlock(X);	$X := X + 1$;
lock(Y);	lock(X);
read(Y);	write(X);
$Y := Y + 1$;	unlock(Y);
write(Y);	unlock(X);
unlock(Y);	

T_1 : $l_1(X), r_1(X), w_1(X), u_1(X), l_1(Y), r_1(Y), w_1(Y), u_1(Y)$: Ok!

T_2 : $l_2(X), l_2(Y), r_2(X), u_2(X), l_2(X), w_2(X), u_2(Y), u_2(Y)$: Ok!

Note: T_2 locks and unlocks X twice and locks Y without a need

Example: Transactions with binary locks 2

<u>T_1</u>	<u>T_2</u>
lock(X);	lock(X);
read(X);	lock(Y);
$X := X - 1$;	read(X);
write(X);	unlock(X);
unlock(X);	$X := X + 1$;
read(Y);	lock(X);
lock(Y);	write(X);
$Y := Y + 1$;	
write(Y);	
unlock(Y);	<u>unlock(X);</u>

T_1 : $l_1(X), r_1(X), w_1(X), u_1(X), r_1(Y), l_1(Y), w_1(Y), u_1(Y)$: Not ok

T_2 : $l_2(X), l_2(Y), r_2(X), u_2(X), l_2(X), w_2(X), u_2(Y)$: Not ok

Note: T_1 reads Y without locking, and T_2 does not unlock Y

Example: Schedules with binary locks

$S_1 : l_1(X), r_1(X), u_1(X), \quad l_2(X), r_2(X), u_2(X), \quad l_1(X), w_1(X), u_1(X)$

Ok!

$S_2 : l_1(X), r_1(X), \quad l_2(X), u_1(X), \quad r_2(X), u_2(X), \quad l_1(X), w_1(X), u_1(X)$

Ok!

$S_3 : l_1(X), r_1(X), \quad l_2(X), r_2(X), u_1(X), \quad u_2(X), \quad l_1(X), w_1(X), u_1(X)$

Not ok (implementation of $l_2(X)$ is 'broken': should not allow to continue T_2 with $r_2(X)$ before $u_1(X)$)

Note: locks (i.e., the rules and implementations as above)
do **not** guarantee serialisability by themselves;
we will see how they can be used to ensure serialisability

Concurrency control subsystem of a DBMS has a lock manager module that relies on a lock table:

- has schema [Data_item_name, Locking_transaction]
- keeps only locked items (others are unlocked)

Lock manager keeps track of and controls access to locks by checking the rules for transactions and enforcing allowed schedules (waiting, etc.)

Binary locking is too restrictive anyway so before moving to the description of how they are used, we consider more advanced locks

Shared/exclusive locks

Shared/exclusive (or read/write) locks **idea**: we can allow **sharing** for several **reading** transactions (but still **exclusive writes**)

Shared/exclusive (or read/write) locks **formally**: for each item X

- LKD(X) is read-locked (or share-locked) means that X is read by a transaction
- **auxiliary numeric** variable $\#READS(X)$ is maintained, which keeps the **number** of reading transactions
- LKD(X) is write-locked (or exclusive-locked) means that X is **exclusively locked** for writing by some transaction and cannot be accessed by others
- LKD(X) is **unlocked** means that X is available for access (with locking)
- schedules start with all LKD(X) = **unlocked** and $\#READS(X) = 0$

Lock table stores this information appropriately
(maintained by **lock manager**)

Shared/exclusive locks transaction operations

Transaction with shared/exclusive locks: can use three operations `read_lock(X)`, `write_lock(X)` and `unlock(X)` with **rules**:

- each `read(X)` is after `read_lock(X)` or `write_lock(X)` (with no `unlock(X)` in between)
- each `write(X)` is after `write_lock(X)` (with no `unlock(X)` in between)
- each `read_lock(X)` and `write_lock(X)` has `unlock(X)` at some point after
- no **changing** of lock type before unlocking; that is, no `read_lock(X)` after `write_lock(X)` without `unlock(X)` between
(this may be changed, see **lock conversion** below)

Implementation of read_lock

The routine that the transaction manager runs when it receives `read_lock(X)` from a transaction:

```
read_lock(X):  
B: if LKD(X) = "unlocked"  
    then begin LKD(X) ← "read-locked";  
           #READS(X) ← 1  
    end  
else if LKD(X) = "read-locked"  
    then #READS(X) ← #READS(X) + 1  
else begin  
    wait (until LKD(X) = "unlocked"  
          and the lock manager wakes up the transaction);  
    go to B  
    end;
```

Implementation of write_lock

The routine that the transaction manager runs when it receives `write_lock(X)` from a transaction:

```
write_lock(X):  
B: if LKD(X) = "unlocked"  
    then LKD(X) ← "write-locked"  
    else begin  
        wait (until LKD(X) = "unlocked"  
            and the lock manager wakes up the transaction);  
        go to B  
    end;
```

Implementation of unlock

The routine that the transaction manager runs when it receives `unlock(X)` from a transaction:

```
unlock (X):  
  if LKD(X) = "write-locked"  
    then begin LKD(X) ← "unlocked";  
           wakeup one of the waiting transactions, if any  
    end  
  else if LKD(X) = "read-locked"  
    then begin  
           #READS(X) ← #READS(X) - 1;  
           if #READS(X) = 0  
             then begin LKD(X) = "unlocked";  
                    wakeup one of the waiting transactions, if any  
             end  
    end;
```

Examples: Schedules with shared/exclusive locks

$S_1 : r_{l_1}(X), r_1(X), u_1(X), \quad r_{l_2}(X), r_2(X), u_2(X), \quad w_{l_1}(X), w_1(X), u_1(X)$

Ok!

$S'_1 : r_{l_1}(X), r_1(X), u_1(X), \quad r_{l_2}(X), r_2(X), u_2(X), \quad r_{l_1}(X), w_1(X), u_1(X)$

Not ok (write with read lock)

$S''_1 : w_{l_1}(X), r_1(X), u_1(X), \quad r_{l_2}(X), r_2(X), u_2(X), \quad w_{l_1}(X), w_1(X), u_1(X)$

Ok!

$S_3 : r_{l_1}(X), r_1(X), \quad r_{l_2}(X), r_2(X), u_1(X), \quad u_2(X), \quad w_{l_1}(X), w_1(X), u_1(X)$

Ok!

$S'_3 : r_{l_1}(X), r_1(X), \quad w_{l_2}(X), w_2(X), u_1(X), \quad u_2(X), \quad w_{l_1}(X), w_1(X), u_1(X)$

Not ok (implementation of $w_{l_2}(X)$ is 'broken': should be exclusive)

$S''_3 : r_{l_1}(X), r_1(X), \quad r_{l_2}(X), r_2(X), \quad u_2(X), \quad w_{l_1}(X), w_1(X), u_1(X)$

Not ok (T_1 is not following the rules: a second lock without unlocking)

Transaction rule

- no changing of lock type before unlocking
(e.g., no `read_lock(X)` after `write_lock(X)` without `unlock(X)` between)

can be **dropped**, thus allowing for **lock conversion**—that is, lock upgrade and downgrade

This may be obviously beneficial for **concurrency**

Implementations of `read_lock(X)`, `write_lock(X)`, and `unlock(X)` need modifications (left for **homework**)

Example: Shared/exclusive locks with conversions

All above are the same, but

$S_3'' : r1_1(X), r_1(X), \quad r1_2(X), r_2(X), \quad u_2(X), \quad w1_1(X), w_1(X), u_1(X)$

Now ok!

Still, do not guarantee serialisability by themselves,
but are needed to guarantee it in the protocols below

1.2. Two-phase locking protocol

Two-phase locking: definition

Conversion-free transaction (binary or shared/exclusive) follows two-phase locking protocol if it follows the **rule**

- all locking operations are **before all unlocking** operations

Two phases:

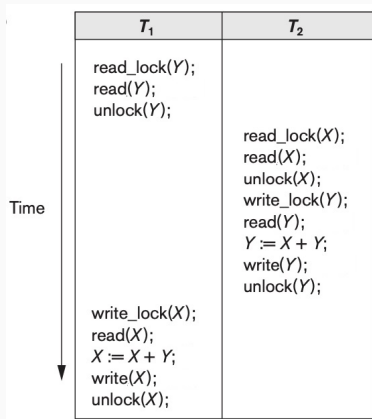
- locking (first, expanding, or growing) phase
- unlocking (second or shrinking) phase

If **conversion** is allowed,

- all upgrades (from read- to write-lock) in **growing phase**
- all downgrades (from write- to read-lock) in **shrinking phase**

Two-phase locking: negative example

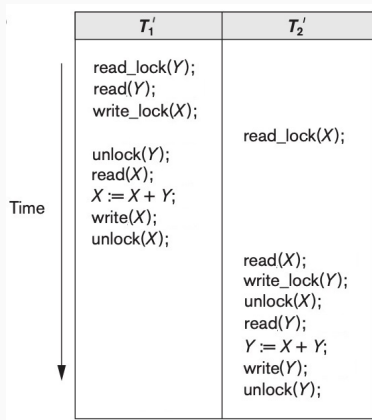
T_1	T_2
<pre>read_lock(Y); read(Y); unlock(Y); write_lock(X); read(X); X := X + Y; write(X); unlock(X);</pre>	<pre>read_lock(X); read(X); unlock(X); write_lock(Y); read(Y); Y := X + Y; write(Y); unlock(Y);</pre>



Transactions T_1 and T_2 (left) do **not** follow the **two-phase locking protocol**, so they allow for a **non-serialisable** schedule (right)

Two-phase locking: positive example

T_1'	T_2'
<code>read_lock(Y);</code> <code>read(Y);</code> <code>write_lock(X);</code> <code>unlock(Y);</code> <code>read(X);</code> <code>X := X + Y;</code> <code>write(X);</code> <code>unlock(X);</code>	<code>read_lock(X);</code> <code>read(X);</code> <code>write_lock(Y);</code> <code>unlock(X);</code> <code>read(Y);</code> <code>Y := X + Y;</code> <code>write(Y);</code> <code>unlock(Y);</code>



Versions T_1' and T_2' (left) follow the **two-phase locking protocol**, so no **non-serialisable** schedule complies the rules

Serialisability by two-phase locking

Two-phase locking **guarantees** serialisability
(i.e., every **allowed** schedule of transactions following the **rules** of
the two-phase locking protocol is **(conflict-)serialisable**)

As the result, we can create a serialisable schedule **operation** by
operation, without knowing the transactions **in full**

Drawback 1: Some serialisable schedules **do not comply** the
protocol, thus **limiting** the amount of concurrency

(formally, there is a conflict-serialisable schedule without locks that cannot be
converted to an **allowed schedule** of transactions following the protocol by
inserting locking and unlocking operations)

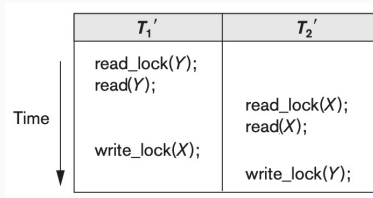
Homework: come up to an example

Serialisability by two-phase locking

Two-phase locking **guarantees** serialisability
(i.e., every **allowed** schedule of transactions following the **rules** of
the two-phase locking protocol is **(conflict-)serialisable**)

As the result, we can create a serialisable schedule **operation** by
operation, without knowing the transactions **in full**

Drawback 2: we may end up in a **deadlock**



We will look at **methods** to deal with **deadlocks** in few minutes

Variations of two-phase locking

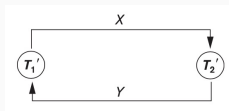
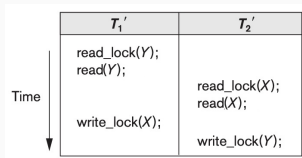
More **constrained** versions of two-phase locking often guarantee **better properties** (while **sacrificing** even more concurrency):

- **conservative** (or **static**) two-phase locking:
all **locking** of a transaction are at its **beginning**
deadlock-free, requires to know the read and write sets at the beginning
- **strict** two-phase locking:
all **write unlocking** of a transaction are at its **end**
not deadlock-free, used in practice, guarantees strict schedules
- **rigorous** two-phase locking:
all **unlocking** of a transaction are at its **end**
not deadlock-free, easy to implement, guarantees strict schedules,
very limited concurrency

1.3. Deadlocks

Deadlock problem

Two (or more) transactions may be stuck in a **deadlock**, where **each** transaction is **waiting** for another transaction, making a 'waiting cycle'



This may happen for **all types** of locks,
so we **do not** assume a specific type below

Deadlock avoidance: Overview

We will consider **several groups of (sub-)protocols** to deal with deadlocks:

- (**naive**)
- deadlock prevention
- deadlock detection
- timeouts

Most of protocols may **roll back** (i.e., abort and restart) some transactions

- **cascadeless** (i.e., **recoverable**) due to two-phase locking (all deadlocked transactions in the growing phase, so no one have read what they have written)
- may face and have to deal with **starvation**:
a transaction rolls back over and over again **indefinitely**

Deadlock avoidance protocols: naive approaches

We have seen one method:

conservative two-phase locking (not practical)

Another simple method:

locking according to a predefined order of all items

- all items are ordered in advance (e.g., by address)
- all locking of a transaction is in the increasing order
- no deadlocks by construction
- not practical due to
 - limited concurrency
 - unreasonable restrictions on transactions

Deadlock prevention protocols: Overview

We will consider **four** more methods for **deadlock prevention**:

- timestamp-based (wait-die and wound-wait)
- waiting-based (no-waiting and cautious-waiting)

All four **avoid deadlocks**, but often **force** transaction **aborts** without a **real deadlock**

Deadlock prevention

is **not** a popular approach in practice due to **high overhead**, but may be useful when transactions are **long** and **use many items**

Timestamp-based deadlock prevention: Wait-die rule

Transaction timestamp:

a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before (for the first time)

Wait-die rule:

let transaction T try to lock an item that is already locked by T'

- if $TS(T) < TS(T')$ (i.e., T is older than T'), then T waits for T' to release
- otherwise (i.e., T is younger) T aborts (i.e., 'dies') and restarts with the same timestamp

The older proceeds (maybe, after waiting) and the younger restarts, so the wait-die rule guarantees no deadlocks, no starvation

Timestamp-based deadlock prevention: Wound-wait rule

Transaction timestamp:

a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before (for the first time)

Wound-wait rule:

let transaction T try to lock an item that is already locked by T'

- if $TS(T) < TS(T')$ (i.e., T is older than T'), then T' aborts (i.e., is 'wounded') and restarts with the same timestamp
- otherwise (i.e., T is younger) T waits for T' to release

The older proceeds and the younger restarts (maybe, after waiting), so the wound-wait rule guarantees no deadlocks, no starvation

Examples of wait-die and wound-wait

When T_1 tries to lock X that is already locked by T_2

- if T_1 is older
 - **wait-die**: T_1 waits
 $b_1, \dots, b_2, \dots, l_2(X), \dots, l_1(X), [T_1 \text{ waits}] \dots, u_2(X), \dots$
 - **wound-wait**: T_2 dies
 $b_1, \dots, b_2, \dots, l_2(X), \dots, l_1(X), a_2, \dots$
- if T_1 is younger
 - **wait-die**: T_1 dies
 $b_2, \dots, b_1, \dots, l_2(X), \dots, l_1(X), a_1, \dots$
 - **wound-wait**: T_1 waits
 $b_2, \dots, b_1, \dots, l_2(X), \dots, l_1(X), [T_1 \text{ waits}] \dots, u_2(X), \dots$

Waiting-based deadlock prevention protocols

(No timestamps in these protocols)

Let transaction T try to lock an item that is already locked by T'

- no-waiting rule:
 - T aborts and restarts
- cautious-waiting rule:
 - if T' is not waiting for anyone, then T waits (regularly checked)
 - otherwise T aborts and restarts

Any of these rules guarantees no deadlocks
(no cyclic waiting is possible)

Special care should be taken to avoid starvation
(Homework: What can be a solution?)

Easier, but even more unnecessary aborts

Deadlock detection

Deadlock happens when
there is a cycle of transactions waiting for each other

We can detect a cycle and
break the cycle by aborting one transaction (the victim)

Can be done by maintaining the wait-for graph:

- nodes: active transactions
- (directed) edges: transactions waiting for transactions

A cycle in the wait-for graph means a deadlock

May be resource-consuming to maintain

Victim selection should be done cautiously to avoid starvation

Timeout-based deadlock avoidance:

fix a waiting **timeout period** in advance

a transaction waits for at most the **timeout period**

after this **abort** and **restart**

Simple and **easy** to maintain

No **guarantees** (unknown overhead, possible starvation)

2. Concurrency via timestamps

Timestamp-based concurrency control: Idea

Reminder:

Transaction timestamp in a unique number $TS(T)$ assigned to each transaction T so that $TS(T') < TS(T)$ for each T' started before

We used timestamps for deadlock prevention in the lock-based concurrency control

We can use timestamps for concurrency control without locking:

- optimistically runs transactions until something goes wrong (i.e., a conflict is detected)
- rolls back one of the conflict transactions based on their timestamps (roll back is abort and restart with a new timestamp)

Timestamp-based concurrency control: Overview

Timestamp-based concurrency control protocols:

- all ensure **serialisability**
 - the **serial version** has transactions arranged by their **timestamps**
- may **not** guarantee **recoverability**
 - additional techniques** needed to ensure this
(e.g., form of **deadlock-free locking**)
- may **not** guarantee no **starvation**
 - additional techniques** needed to ensure this

We will consider several **timestamp-based protocols**

- **basic**: ensures **serialisability**, but **not recoverability**
- **strict**: ensures **serialisability** and **strictness** (so **recoverability**)
- **basic/strict** with **Thomas write rule**: less **roll-backs**

Item timestamp variables

Besides the timestamp of a **transaction** (see above), the concurrency control subsystem (of each timestamp-based protocol) maintains **two variables** for each item X :

- read timestamp $ReadTS(X)$:
the timestamp of the **youngest** transaction that **has read** X
- write timestamp $WriteTS(X)$:
the timestamp of the **youngest** transaction that **has written** X

Maintained in appropriate **timestamp tables**
(similar to the **lock table** in lock-based protocols)

(All **initialised** by 0)

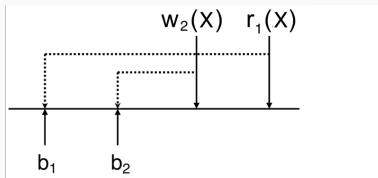
Basic timestamp-ordering protocol: Idea

Main idea:

If there is a **conflict** that violates the order
(of the **serial schedule** imposed by the timestamps),
then **roll back** (i.e., abort and restart with new timestamp)
the transaction of the **second** operation of the conflict

Read-write conflict 1:

a transaction may try to **read** data item X **too late**



Should **abort** and **restart** T_1

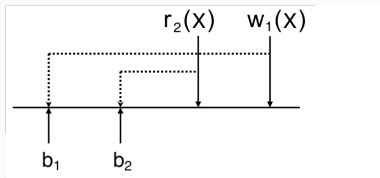
Basic timestamp-ordering protocol: Idea

Main idea:

If there is a **conflict** that violates the order
(of the **serial schedule** imposed by the timestamps),
then **roll back** (i.e., abort and restart with new timestamp)
the transaction of the **second** operation of the conflict

Read-write conflict 2:

a transaction may try to **write** data item X **too late**



Should **abort** and **restart** T_1

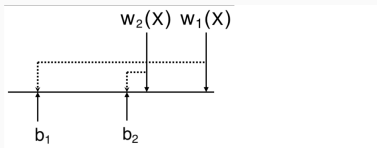
Basic timestamp-ordering protocol: Idea

Main idea:

If there is a **conflict** that violates the order
(of the **serial schedule** imposed by the timestamps),
then **roll back** (i.e., abort and restart with new timestamp)
the transaction of the **second** operation of the conflict

Write-write conflict:

a transaction may try to **write** data item X **too late**



Should **abort** and **restart** T_1 (or?...)

Basic timestamp-ordering protocol formally

Concurrency control subsystem should follow **two rules**:

Rule 1. When transaction T issues $\text{read}(X)$:

- if $\text{WriteTS}(X) > \text{TS}(T)$
(i.e., someone **younger** written X),
then **roll-back** T (i.e., abort and restart T with a new timestamp)
- otherwise, **execute** the rest of $\text{read}(X)$ and
set $\text{ReadTS}(X)$ to $\max(\text{ReadTS}(X), \text{TS}(T))$

Rule 2. When transaction T issues $\text{write}(X)$:

- if $\text{ReadTS}(X) > \text{TS}(T)$ or $\text{WriteTS}(X) > \text{TS}(T)$
(i.e., someone **younger** accessed X),
then **roll-back** T (i.e., abort and restart T with a new timestamp)
- otherwise, **execute** the rest of $\text{write}(X)$ and
set $\text{WriteTS}(X)$ to $\text{TS}(T)$

Basic timestamp-ordering protocol properties

Concurrency control subsystem should follow **two rules**:

Rule 1. When transaction T issues $\text{read}(X)$:

- if $\text{WriteTS}(X) > \text{TS}(T)$ then **roll-back** T
- otherwise, **execute** $\text{read}(X)$ and **set** $\text{ReadTS}(X)$ to $\max(\text{ReadTS}(X), \text{TS}(T))$

Rule 2. When transaction T issues $\text{write}(X)$:

- if $\text{ReadTS}(X) > \text{TS}(T)$ or $\text{WriteTS}(X) > \text{TS}(T)$ then **roll-back** T
- otherwise, **execute** $\text{write}(X)$ and **set** $\text{WriteTS}(X)$ to $\text{TS}(T)$

Ensures **serialisable** schedules, but they

- may be **not recoverable**, (ensured by **strict** version)
- may have **cascading roll-backs**, (avoided by **strict** version)
- may have **starving** transactions (leave for you)

To ensure **durability** in **ACID**, a schedule may be

- **recoverable**: T does not commit before each T' from which T read have committed
- **cascadeless**: T does not read from uncommitted transactions
- **strict**: T is cascadeless and does not overwrite any values written by uncommitted transactions

Strict timestamp-ordering protocol

Idea: In the rules the execution is **postponed** (using e.g., locking) until the pervious write is committed (so ensuring **strict** schedules)

Rule 1. When transaction T issues $\text{read}(X)$:

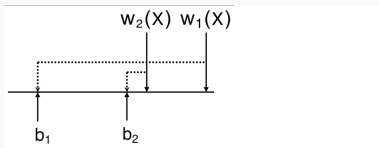
- if $\text{WriteTS}(X) > \text{TS}(T)$ then **roll-back** T
- otherwise
 - **wait** until the transaction T' such that $\text{WriteTS}(X) = \text{TS}(T')$ is **committed** (if there is such)
 - **execute** the rest of $\text{read}(X)$ and **set** $\text{ReadTS}(X)$ to $\max(\text{ReadTS}(X), \text{TS}(T))$

Rule 2. When transaction T issues $\text{write}(X)$:

- if $\text{ReadTS}(X) > \text{TS}(T)$ or $\text{WriteTS}(X) > \text{TS}(T)$ then **roll-back** T
- otherwise
 - **wait** until the transaction T' such that $\text{WriteTS}(X) = \text{TS}(T')$ is **committed** (if there is such)
 - **execute** the rest of $\text{write}(X)$ and **set** $\text{WriteTS}(X)$ to $\text{TS}(T)$

Thomas writing rule

Improvement (less roll-backs) for both basic and strict protocols in case of write-write conflict



Rule 1. When transaction T issues $\text{read}(X)$: ... (same)

Rule 2. When transaction T issues $\text{write}(X)$:

- if $\text{ReadTS}(X) > \text{TS}(T)$ then roll-back T
- otherwise, if $\text{WriteTS}(X) > \text{TS}(T)$ ignore the operation and proceed
- otherwise ... (same, i.e., possibly wait, execute, update)

Same guarantees as the versions without the improvement

3. Multiversioning for concurrency control

Multiversioning concurrency control: Idea

Multiversioning idea:

several **versions** of the (value of the) **same** item are **kept** by the system

- **transactions** are the **same** as usual
(in particular, they request reads/writes for **items**, **not** versions)
- the **versions** of items are managed by the **concurrency control subsystem**

Pros: increase **concurrency** (we will see how)

Cons: more **storage** is needed to maintain multiple versions

Multiversioning for concurrency control: Overview

Multiversioning: several **versions** of the **same** item are kept

Technical difficulty (for us):

Intuitively, still want to ensure **semantic equivalence** to a **serial** schedule, but our notion of a **schedule** does **not** cover **multiple versions** of an item, and all **derived** notions (conflicts, serialisability, etc.) are **not** formally **applicable**

All these notions can be **generalised** appropriately (but **clumsy**)

So we assume such generalisation **silently**

Aiming for **semantic serialisability** of the **multiversion schedule**

We will (briefly) consider **several multiversion protocols**, based on

- timestamps
- two-phase locking
- validation
- snapshot isolation

Timestamps and timestamp variables for multiversioning

Concurrency control subsystem **maintains**:

- a **timestamp** $TS(T)$ of each transaction T (as before)
- several **versions** X_0, X_1, \dots of **each** data item X
each version is written (created) only **once**
- two **variables** for each **version** X_i of each item:
 - **read timestamp** $ReadTS(X_i)$:
the timestamp of the **youngest** transaction that **has read** X_i
 - **write timestamp** $WriteTS(X_i)$:
the timestamp of the (**only**) transaction that **has written** X_i

Maintained in appropriate **tables**

(similar to the **lock** table in lock-based protocols)

(Each X_0 is **initialised** by the **original** value with 0 timestamps, at the end the **last version** is considered **final**)

Multiversion timestamp-ordering protocol

Main idea:

- A transaction reads the version written by the **youngest** transaction among those that are older than the reading transaction (always allowed)
- A transaction is allowed to write (create a new version) **only** if **no one** yet read a **wrong** value

Formally, concurrency control subsystem should follow **two rules**

Rule 1. When transaction T issues `read(X)`:

- find the version X_i with the **highest** $WriteTS(X_i)$ **less or equal** than $TS(T)$ (if there are several versions with the same timestamp, take the one with the **largest** i)
- return **this version** to T and
 set $ReadTS(X_i)$ to $\max(ReadTS(X_i), TS(T))$

Multiversion timestamp-ordering protocol

Main idea:

- A transaction reads the version written by the **youngest** transaction among those that are older than the reading transaction (always allowed)
- A transaction is allowed to write (create a new version) **only** if **no one** yet read a **wrong** value

Formally, concurrency control subsystem should follow **two** rules

Rule 2. When transaction T issues `write(X)`:

- find version X_i with **highest** $WriteTS(X_i)$ **less or equal** than $TS(T)$
- if $ReadTS(X_i) > TS(T)$ (i.e., someone **younger** accessed X),
then **roll-back** T (i.e., abort and restart T with a new timestamp)
- otherwise, create a **new version** X_j and **set**
both $ReadTS(X_j)$ and $WriteTS(X_j)$ to $TS(T)$

Multiversion timestamp-ordering protocol: Properties

Main idea:

- A transaction reads the **youngest** version among those that are created by **older transactions** (always allowed)
- A transaction is **allowed to write** (create a new version) only if **no one** yet read a **wrong** value

Ensures **serialisable** schedules (with the **timestamp-ordered serial** schedule), but they may

- be **not recoverable**, (ensured by **strict** version)
- have **cascading roll-backs**, (avoided by **strict** version)
- have **starving** transactions

The **strict** (hence **recoverable and cascadeless**) version may be designed in the **same way** as for usual timestamp-based protocol

Multiversion two-phase locking protocol: Idea (only)

Maintains **two versions** of each data item:

- **committed**:
 - written by a **committed** transaction (or **original**)
 - all transactions (except the currently writing) **read this**
- **local**:
 - written by a **currently writing** transaction (only **one** is allowed)
 - visible for **reading** only for the **writing transaction**
 - **rewrites** the committed version
when the writing transaction **commits**

Three types of locks (**read**, **write**, **certify**) with compatibility table

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Validation-based protocol: Idea

Validation-based protocol:

- an **optimistic** serialisation strategy based on **time stamping**
(‘**optimistic**’ is ‘evaluate transaction and check serialisability **after**’)
- each transaction has **three phases**:
 - **read** phase (includes **all calculation**)
 - **validation** phase (few **checks**, can **roll back**)
 - **write** phase(equivalent to keeping **multiple local versions** of items)
- the concurrency subsystem stores information only **about transactions** (**not about items**)
- aims for the **serialisable** schedule equivalent to the **serial** one according to the **order of** (completion of) **validation**
- (may be **not recoverable** and **starving**, need separate techniques to avoid them)

Validation-based protocol:

- each transaction has **three phases**:
 - **read** phase:
 - includes **all calculation**
 - **validation** phase:
 - few **checks**, can **roll back**
 - in what follows, we assume that it has only **one** operation
 - **write** phase
- the concurrency subsystem **stores**, for each transaction T :
 - **beginning** timestamp of **each phase**
 $Start(T)$, $StartVal(T)$, $StartWrite(T)$ (passed so far)
 - **end** timestamp $End(T)$ (passed so far)
 - **read** and **write set** of items $ReadSet(T)$ and $WriteSet(T)$
(should be known by $StartVal(T)$)

Validation-based protocol: Formally

At **validation phase** of each transaction T , check the following
for each T' with $StartVal(T') < StartVal(T)$

(i.e., **older** according to the validation order):

– if $End(T') < Start(T)$ (i.e., T' ended before T started), then **pass**

– otherwise,

if $End(T') < StartVal(T)$ (i.e., T' ended before T writes), then

$$WriteSet(T') \cap ReadSet(T) = \emptyset$$

(i.e., T' wrote nothing T has read, maybe too late)

– otherwise (i.e., T' started to write before T writes), then

$$WriteSet(T') \cap (ReadSet(T) \cup WriteSet(T)) = \emptyset$$

(i.e., T' wrote nothing T is accessing, maybe too late)

If a check fails, **abort** T

Ensures **serialisability** according to the **validation order**

Snapshot isolation: Idea

Snapshot Isolation concurrency control protocol

- used in several **practical** DBMSs (PostgreSQL, Oracle, etc.)
- **easy to ensure** for the manager
- each transaction evaluates on the data **version** committed by the **beginning** of the transaction (i.e., **multiversion**)
- **write-sets** of interleaving transactions are **disjoint**
- does **not** guarantee **serialisability**

Example: $r_1(X), r_1(Y), r_2(X), r_2(Y), w_1(Y), w_2(X), c_1, c_2$

On the one hand, **not** (conflict-)serialisable

On the other hand, **snapshot-isolated** in essence:

- both transactions read only **initial** data
- **write-sets** are **disjoint**

4. Concurrency with multiple granularity

Data item granularity

Granularity: the size of data items

- field value of a record (**fine**)
- record
- disk block
- table file
- whole database (**coarse**)

The **tradeoff**:

- more coarse allows for **lower** degree of **concurrency**
- more fine leads to **more overhead** (lock management, etc.)

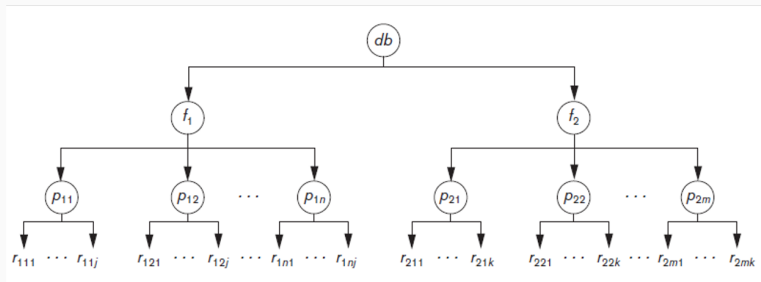
Best item size depends on **transaction type**:

- if a transaction accesses **few** records, a **record** as item is good
- if a transaction accesses **many** records, a **block** as item is good

It may be nice to have **different granularity** for different transactions

Multiple granularity locking: Example

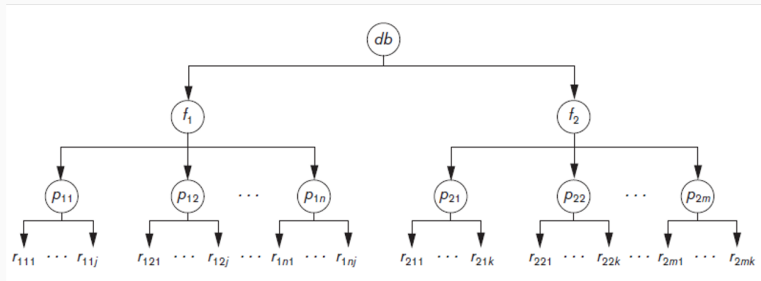
Example granularity hierarchy:



Let T_1 write-lock f_1 and then T_2 tries to read-lock r_{121} :
easy to detect and delay T_2 by traversing from leaf to root

Multiple granularity locking: Example

Example granularity hierarchy:



Let T_2 read-lock r_{121} and then T_1 tries to write-lock f_1 :
detecting requires to search through the whole sub-tree,
which may be **not efficient**

Intention locks

Intention locks solve the problem:

besides usual (exclusive or shared) locks,
the locking transaction labels the path
from the root to the needed node with intention locks

- Intention-shared lock (IS):
a shared lock is to be requested on a descendant node
(one or several)
- Intention-exclusive lock (IX):
an exclusive lock is to be requested on a descendant node
- Shared-intention-exclusive lock (SIX):
current node is shared-locked and
an exclusive lock is to be requested on a descendant node

Multiple-granularity two-phase locking

Multiple-granularity two-phase locking protocol rules (informally):

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

- each **locking** starts in the **root** and ends in the **locked node**
- the **path** to the node is locked with **intentional versions** and the **node** itself with the **usual** lock
- the lock-compatibility **table** (above) must be always adhered (**wait if 'no'**)
- **unlocking** is symmetric, **upgrading** is possible

5. Other issues and conclusion

There are **many more issues** with concurrency control:

- **indexes**, in particular $B(+)$ -trees
 - using **multiple-granularity two-phase locking** is possible
 - not very efficient**: insertion write-locks all the branch
 - dedicated** approaches are used
 - (using the fact that all insertions start from **the root**)
- **insertion** and **deletion** may require special care
 - (write locks, etc.)
- **phantom tuples** may require special treatment
 - (when a new tuple is **inserted** in the middle of an **affected selection** by another transaction)
 - our protocols **do not** cover them
- **interactive transactions**
 - (something showed to a user, but then should be rolled back)

What have we learned?

Concurrency control **protocols**

that ensure **isolation** via (conflict-)serialisability:

- **locking-based**

 - (two-phase with variations,
based on binary and shared/exclusive locks
with several deadlock-avoidance methods)

- **timestemp-based**

 - (basic, strict, Thomas write rule)

- **multiversion-based**

 - (timestamp, two-phase, validation, snapshot)

- **multiple-granularity-based**

In the rest of **Part 2** (next week):

- techniques for **recovery**

IN3020&4020 – Database Systems (2024)

Part 2: Transaction management

Lectures 22, 23: Recovery techniques

9, 15 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓

Part 1. Query processing in relational databases ✓

Part 2. Transaction management in relational databases

- Transaction processing concepts (3 lectures) ✓
- Concurrency control techniques (3 lectures) ✓
- Database recovery techniques (2 lectures)

Part 3. NoSQL DBMS

Part 2. **Transaction management in relational databases**

Fundamentals of Database Systems by R. Elmasri and S.B. Navathe (7th edition): **Part 9** (close to my presentation)

- Transaction processing concepts (3 lectures)

Chapter 20

- Concurrency control techniques (3 lectures)

Chapter 21

- **Database recovery techniques** (2 lectures)

Chapter 22

Database Systems: the Complete Book by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): **Parts of Section 17**

etc.

Transactions: Reminder

Transaction is a sequence of **operations** (not arbitrary)

- **begin**, **end**, **commit**,
abort, **read(X)**, **write(X)**, variable updates, etc.
- we use **abbreviations** (**b**, **e**, **c**, **a**, **r(X)**, **w(X)**)
- we may **omit** operations (e.g., use only **r(X)** and **w(X)**)
- we often append transaction **ids** (e.g., **b₁**, **r₁(X)**, **r₂(X)**, **w₃(X)**)
- transactions read from and write to the common disk (or **cache!**),
but have their own main memory

Can be **committed**, **aborted** or **partial**

Schedule (history, or **execution plan**) S of transactions T_1, \dots, T_n is a (total) **ordering** of the operations of T_1, \dots, T_n

- operations of different transitions can interleave
- operations of each T_i are in the same order as in T_i

Complete schedule: all T_1, \dots, T_n are **committed** or **aborted**

ACID principles: Reminder

ACID principles:

- **Atomicity**: Transaction performed in its entirety or not at all
Ensured by the **recovery subsystem**
- **Consistency**: The database should always remain consistent
Ensured by the transaction program ✓
- **Isolation**: Transaction should not interfere with others
Ensured by the concurrency control subsystem ✓
- **Durability**: Changes of committed transactions must persist
Ensured by the **recovery subsystem**

In general, the subsystems need **coordination**

Recoverable Transactions: Reminder

To ensure **durability** in **ACID**, a schedule may be

- **recoverable**: T does not commit before each T' from which T read have committed
- **cascadeless**: T does not read from uncommitted transactions
- **strict**: T is cascadeless and does not overwrite any values written by uncommitted transactions

In this part, we concentrate on **strict** schedules
(restricted concurrency, but enough to illustrate the principles)

1. Recovery concepts
 - 1.1. Intro
 - 1.2. Cache
 - 1.3. Log
 - 1.4. Checkpoint

2. Recovery protocols
 - 2.1. Shadowing recovery (no logs)
 - 2.2. Deferred-update recovery (NO-UNDO/REDO logs)
 - 2.3. Immediate-update recovery (UNDO/REDO logs)
 - 2.4. ARIES recovery (advanced UNDO/REDO logs)

3. Other considerations

1. Recovery concepts

1.1. Intro

Recovery:

restores database to most recent consistent state before **failure**

Types of failures (reminder):

- Computer failure (system crash): hardware, software, network errors
- Transaction failure: division by zero, constraint violation, etc.
- Local transaction errors: no data found, programmed exception, etc.
- Concurrency control enforced: serialisability violation, deadlock break, etc.
- Disk crash: persistent errors with disk reads or writes
- Physical problems: power cut, fire, catastrophe, etc.

Groups of failures:

- Disk is (significantly) **damaged** (catastrophe, disk crash, etc.)
- Disk is **not damaged** (everything else)

Recovery:

restores database to most recent consistent state before **failure**

Groups of failures:

- **Disk** is (significantly) **damaged** (catastrophe, disk crash, etc.)
restore the **whole** database from **back-up storage** (tape, etc.)
few words at the end of this slides
- **Disk** is **not damaged** (everything else)
recover consistency by **undoing** and **redoing**
some operations following a **recovery protocol**
use the database on the **disk** and a **log of operations**
need to take into account the **cache** in the main memory
our **main focus**

1.2. Cache

DBMSs use (dedicated) **cache**:

- some parts of information (**data itself, indexes, logs**, etc.) from the disk are kept in main-memory **cache** for quick access
- usually, collection of **buffers** (a.k.a. **pages**—that is, blocks in **main memory**) with corresponding blocks on **disk**
- to perform an **operation** with data, the **cache** is first checked:
 - if the needed block is in **cache**, it is **used**
 - otherwise, an existing cache buffer is **flushed** (i.e., **replaced**) by the needed block from disk using **buffer replacement policies** (i.e., methods to decide which cache buffers should be flushed)
 - if the operation is **write**, then the buffer becomes **dirty**: its disk version is outdated

Cache: Flushing approaches

Each buffer in the cache has a **dirty bit** (flag) attached:

- was the buffer **modified** or **not**

Flushing approaches:

- **in-place** flushing:
 - new (i.e., cache) version **replaces** the old (the disk) one
 - if dirty bit is 1
 - used in **most cases**
- **shadowing**:
 - new version is written in **another place** and
 - the old version is kept
 - no log** is needed for recovery (see below), but **overhead**
 - used in the **shadowing recovery** protocol (see below)

Cache: Buffer replacement policies

DBMSs use **cache**: some disk blocks are mirrored in cache (main memory) buffers for quick access, but sometimes we have to **replace** a cache buffer with a new one

Basic standard replacement policy is **LRU**:

- the **least recently used** buffer is flushed (replaced)
- not specialised for DBs and not effective, because different domains (types of information: **data itself, indexes, logs**, etc.) may benefit from different treatment

Cache: Buffer replacement policies

DBMSs use **cache**: some disk blocks are mirrored in cache (main memory) buffers for quick access, but sometimes we have to **replace** a cache buffer with a new one

Domain separation technique:

- Each **domain** (data itself, indexes, logs, or even certain relations) has its **own** dedicated cache
- **LRU** (least recently used) policy is used in each **individually**
- better results than common LRU

Can be **improved** in several ways:

- **Group LRU** with **dynamic** cache separation, **DBMIN** (omitted)
- **Hot set, clock sweep** (next slide)

Cache: Clock sweep with hot set

Hot set:

- blocks that should **always** be in cache until a certain point
- decided **externally** (e.g., smaller relation in **nested-loop join**)

Clock sweep over other buffers (for each domain **separately**):

- each (non-hot) cache buffer has an integer **count value**
- cache buffers are checked in a **round-robin 'cycle'**
 - with some **regularity**, maintaining **individual counts**
- if a buffer has started to be used since the last time visited, **increment** its count
- otherwise, **decrement** its count
- if we need to **replace** a buffer with a new block from the disk, the buffer with the **least count** is replaced

1.3. System Log

System log keeps track of **executed operations** of transactions:

- **sequential, append-only** file (i.e., a table of entries)
- reading and writing the log are **not** affected by **failure** except disk or catastrophic failures
(i.e., logging is done by stand-alone system operations which is not a part of any transaction operations)
- **backed up** periodically to guard against these failures

System log: Properties

System log keeps track of **relevant** executed operations of transactions:

- in particular, we do **not** need to log **computations**
- log keeps the **effects** of data writes and maybe reads
(not write and read operations themselves,
see next slide)
- log also keeps the **beginnings** and the **ends** (commit or abort)
of transactions
(e.g., entries of the form `[commit, T]`)
- log keeps **other information**
e.g., about **checkpoints** and **dirty buffer** table
(will discuss when needed)

System log: Read and Write entries

System log table can have **read** and **write** entries of several types:

- **UNDO- (OLD-) WRITE** entry:
 - keeps the **before-image** (i.e., **old value**) of a written item used to **undo** the write if the transaction is aborted
- **REDO- (NEW-) WRITE** entry:
 - keeps the **after-image** (i.e., **new value**) of a written item used for **redo** the write if the transaction is committed, but the effect is not on the disk (only in a dirty cache buffer)
- **UNDO&REDO-WRITE** entry: keeps both above
- additionally, **READ** entry:
 - needed when **cascading rollbacks** are possible
 - we generally assume **strict schedules**, so do not use such entries
 - (reminder: a schedule is **strict** when no transaction can read or write an item X until the previous write of X is committed or aborted)

System log cache buffer:

- main memory buffer (cache)
- keeps the last part of the log
- when full, appended (i.e., flushed) to the log file on the disk

Log cache buffer may also be flushed to ensure write-ahead logging:

- log information about a transaction operation is flushed before the effect of the operation is flushed itself

Write-ahead logging: More detail

A recovery protocol follows **write-ahead logging** if

- **before-image** (the **old version**) of each item on the disk **cannot be overwritten** (by the **flushing** buffer) by its **after-image** (the **new version**) until **all UNDO (OLD)** log entries have been **flushed** to disk
- a transaction **cannot commit** until all **UNDO (OLD)** and **REDO (NEW)** log entries for that transaction have been **flushed** to disk

(Most of) recovery protocols are **write-ahead logging**

1.4. Checkpoints

Checkpoints

Checkpoint idea:

a recovery subsystem (not transaction!) **operation** that makes it possible, in case of a **fail**, to safely **return** to a consistent state (i.e., all committed transactions at a point of the fail are logically redone, all aborted undone) by

- **flushing** all relevant buffers and
- logging a **list of currently active transactions**

Ensures that **everything committed so far** is on the disk and **does not** need any action in case of a fail

Repeated regularly (e.g., every 5 minutes) by **recovery subsystem**

- **more** frequent—**less** recovery work
- **less** frequent—**more** overhead

The latest checkpoint is **active**

Checkpoints

Checkpoint idea:

a recovery subsystem (not transaction!) **operation** that makes it possible, in case of a **fail**, to safely **return** to a consistent state (i.e., all committed transactions at a point of the fail are logically redone, all aborted undone) by

- **flushing** all relevant buffers and
- logging a **list of currently active transactions**

Basic (rigid) **checkpoint** operation formally:

1. **suspend** execution of all transactions
2. **flush all** main non-pinned dirty memory buffers
3. **write** checkpoint entry [**checkpoint**, *list of active transactions*]
to log and **flush** the log to the disk
4. **resume** executing transactions

2. Recovery protocols

Coordination of recovery and concurrency control

Recovery subsystem ensures **atomicity** and **durability**:

restores database to most recent consistent state before **failure**—
that is, one or several (maybe all) transactions **abort** and
all their changes need to be **rolled back**

Needs **coordination** with **concurrency control**:

- usually, we assume a **system crash** event that **aborts all** transactions as a **model** type failure
(but the ideas applicable to **other types** of failures,
including those that affect only one transaction)
- usually, we assume **strict two-phase locking** as a **model** concurrency control protocol
(but the ideas applicable to **other concurrency protocols**;
a difficult case is **cascading** roll-backs,
which require more **complicated** treatment)

2.1. Shadowing recovery protocol

Shadowing recovery protocol: Idea and properties

Shadowing (shadow paging) recovery protocol idea:

- flush each modified buffer **immediately** to a **new** place on disk, the old (shadow) version is **kept** on disk (and **never** modified)
- if a transaction **commits**, the **current** version is used
- if a transaction **aborts**, the **shadow** version is used

Shadowing recovery protocol properties:

- **no log** required in **strict** (or single-user) environment (assuming blocks as data items)
- so, sometimes called **NO-UNDO/NO-REDO**
- **no checkpoints** either
- essentially, a **copy** of cache on disk
- results in a lot of **overhead** and **random-ordered** blocks on disk
- requires **complicated** garbage collection

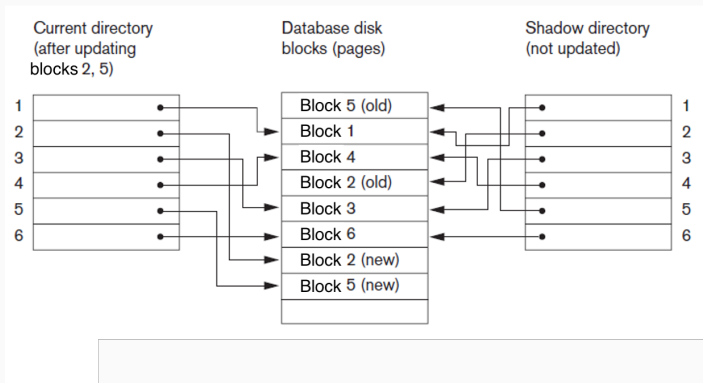
Shadowing recovery protocol: Formally

A directory: a table with pointers to disk blocks

Shadowing (shadow paging) recovery protocol
(for a single transaction):

- at the beginning of a transaction,
a current directory with pointers to all blocks is created,
and it is copied to shadow directory
- after each write,
affected buffers are immediately flushed to a new disk block,
and the current directory is updated
- if a transaction commits, the current version is taken
- if a transaction aborts, the shadow version is restored

Example of shadowing

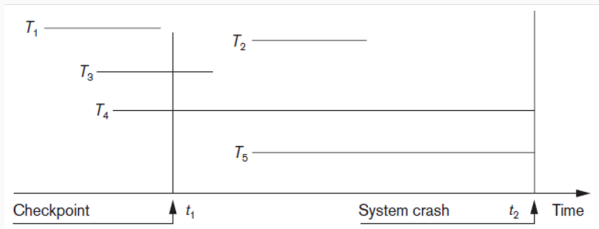


Directories and disk blocks

after writing in Block (i.e., page) 2 and then Block 5

2.2. Deferred-update protocol

Before we start: A picture to have in the head



Here, T_1, \dots, T_5 lines are transactions (committed or partial) spanning over time

Deferred update: Idea

Deferred update recovery protocol idea:

- **no-steal** approach:
 - postpone all flushes of (dirty buffers) to disk until the transaction commits
- **REDO-** (NEW-) log entries are **needed**:
 - to recover committed transactions after a **system crash**
- **UNDO-** (OLD-) log entries are **not needed**:
 - no changes of aborted transactions are on the disk
- thus, **NO-UNDO/REDO**

Deferred update recovery protocol properties:

- effective only for **short transactions** with **few changes**
- **cache size** is an issue with longer transactions

Simplifying **assumptions**, some already mentioned

(can be relaxed but rules and procedures may need adjustments):

- **strict** two-phase locking with **blocks** as items
- system **crash** as a fail (i.e., **all** active transactions abort)
- all blocks changed by a transaction fit into cache

Deferred update protocol rules:

- use **log** with REDO entries and regular **checkpoints**
- all buffers changed by a transaction are **pinned** until commit
(i.e., **no-steal** approach)
- transaction does **not commit** until all its **REDO-type** log entries are recorded in log and **log buffer** is flushed to disk
(i.e., the relevant **REDO-part** of the write-ahead approach)

Deferred-update procedure

(evaluated by the **recovery subsystem** after a system crash):

- find the **active** (i.e., latest) checkpoint in the log on the disk
 - all effects of committed transactions are on the disk by the checkpoint
- using the checkpoint's active transactions and the log after the checkpoint, construct the set of **all committed** transactions between the checkpoint and the crash
 - their commit log entries are on the disk
- **redo** all operations of the committed transactions from the set
 - all their log entries are on the disk

Deferred update: Observations

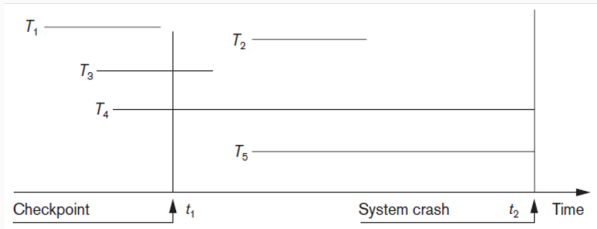
Rules (brief reminder):

- no flushing of modified data before commit
- flushing of all transaction-relevant log entries before commit

Observations:

- ensures **Atomicity** and **Durability** as required
- No need to do anything with **non-committed** transactions
(due to deferred update)
- The steps can be **optimised** and **parallelised**
(e.g., only the last update of each buffer needs to be redone)
- **Restart** the same recovery procedure in case of **another crash**
during recovery

Deferred update protocol: Example



- T_1 : already committed and flushed, no action required
- T_2 : may have non-flushed buffers, but the log (including `begin_transaction`) is flushed, so all changes can be recovered from the log (considered from from the checkpoint)
- T_3 : may have non-flushed buffers, but the log is flushed and the checkpoint remembers that it is active, so all the changes can be recovered from the log
- T_4, T_5 : aborted and no buffers are flushed, no action required

2.3. Immediate-update recovery

Immediate update: Idea

Immediate update recovery protocol idea:

- possible **steal** approach:
 - allow to flush of dirty buffers
 - both before and after the transaction commits
- **REDO-** (NEW-) log entries log entries are **needed**:
 - to redo committed transactions after a **system crash**
- **UNDO-** (OLD-) log entries log entries are **needed**:
 - to undo aborted transactions
- thus, **UNDO/REDO** (generalises deferred-update)

(UNDO/NO-REDO also exists, think about it by yourself)

Immediate update recovery protocol properties:

- more **general** than **deferred-update**
- more **difficult**

Immediate update: Formal rules

Simplifying **assumptions**, less than above

(can be relaxed but rules and procedures may need adjustments):

- **strict** two-phase locking with **blocks** as items (not essential here)
- system **crash** as a fail (i.e., all active transactions abort)

Immediate update protocol rules (just write-ahead logging in full):

- use **log** with REDO/UNDO entries and regular **checkpoints**
- a dirty buffer **does not** flush until all relevant log entries are recorded in log and **log buffer** is flushed to disk
(i.e., the first part of the write-ahead approach)
- transaction **does not** commit until all its log entries are recorded in log and log buffer is flushed to disk
(i.e., the second part of the write-ahead approach)

Immediate update: Recovery procedure

Immediate-update procedure

(evaluated by the recovery subsystem in case of a crash):

- find the **active** (i.e., latest) checkpoint in the log on the disk
all effects of committed transactions are on the disk by the checkpoint
- construct the sets of all **committed** and **non-committed** (i.e., aborted) transactions between the checkpoint and the crash
their commit log entries are on the disk
- **redo** all operations of the committed transactions
all their log entries are on the disk
- **undo** all known operations of the non-committed transactions
some of their log entries may not be on the disk,
but their corresponding changes were not flushed

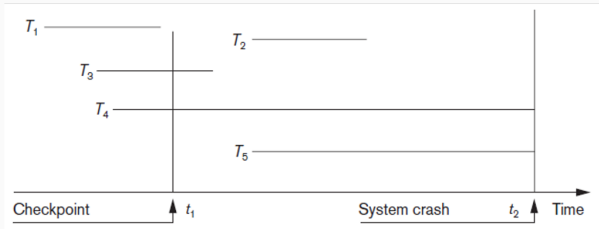
Rules (brief reminder, write-ahead logging):

- flushing of all modified-data-relevant log entries before flushing this modified data
- flushing of all transaction-relevant log entries before commit

Observations:

- ensures **Atomicity** and **Durability** as required
- The steps can be **optimised** and **parallelised**
(e.g., only the last update of each buffer needs to be redone)
- **Restart** the same recovery procedure in case of **another crash** during recovery

Immediate update protocol: Example



- T_1, T_2, T_3 : same as before
- T_4, T_5 : aborted, so should be undone;
some changes may be already flushed by t_2 and some may not,
but the log entries of all flushed changes are also flushed

Improvement for both protocols: Fuzzy checkpoints

Basic (rigid) `checkpoint` reminder:

1. `suspend` execution of all transactions
2. `flush all` main non-pinned dirty memory buffers
3. `write` checkpoint entry to log and `flush` the log to the disk
4. `resume` executing transactions

Step 2 may take `time`,

and suspending for this time all execution may be `impractical`

Fuzzy `checkpoint` allows to `continue` transactions while flushing:

1. `log` begin checkpoint entry [`begin_checkpoint`, *active transactions*]
2. `flush all` main non-pinned dirty memory buffers
 `continuing` transaction execution `in parallel`
3. `log` end checkpoint entry [`end_checkpoint`]
 and `flush` the log to the disk

Checkpoint becomes active after `end_checkpoint`,

but recovery process starts from `begin_checkpoint`

5. ARIES recovery protocol

ARIES protocol main properties:

- Algorithm for Recovery and Isolation Exploiting Semantics
- an improved version of immediate-update protocol
(possible steals, REDO/UNDO log entries, write-ahead logging)
- used in practice
- again, we concentrate on strict schedules

ARIES update recovery protocol improvement ideas:

- different checkpoints (main improvement):
instead of flushing everything,
remember the table of currently dirty buffers
- other improvements (e.g., logging of recovery):
not discussed here

ARIES Checkpoints

During execution, ARIES **maintains** (in main memory):

- **Transaction table** of currently **active** and committed transactions
(ID, pointer to the most recent relevant log entry, status)
- **Dirty buffer table** of currently **dirty buffers** in the cache
(ID, pointer to the earliest update log entry)

ARIES checkpoint (**fuzzy** in general)

- the **begin_checkpoint** entry identifies the start point for the recovery, and the state of the transaction table and dirty buffer table to store
- the **end_checkpoint** entry has the tables (at the begin-point state) **appended** in the log
- the log must be **flushed** at the end of checkpoint

ARIES recovery protocol: Example

(a)

Lsn	Last_lsn	Tran_id	Type	Block_id	Other_information
1	0	T_1	write	C	...
2	0	T_2	write	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	write	A	...
7	2	T_2	write	C	...
8	7	T_2	commit		...

(b)

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	2	in progress

Block_id	Lsn
C	1

(c)

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	8	committed
T_3	6	in progress

Block_id	Lsn
C	7

- (a) The log
- (b) Possible transaction and dirty block table at the begin checkpoint (appended to `end_checkpoint`)
- (c) Possible tables at the moment of the end of the log (observe that 7 in dirty buffer table implies that C was flushed between Lsn 5 and 7)

ARIES recovery main steps

ARIES recovery procedure (evaluates at the crash event):

1. **Analysis step:**

identifies the earliest update log entry of a dirty buffer at the checkpoint (and collects other relevant information)

2. **Redo step:**

go through the log from the identified point to the end and redo the necessary operations (i.e., the writes of dirty blocks in the table before the begin checkpoint and all writes after it)

3. **Undo step:**

go through the log from the end to the beginning of all uncommitted (by the log end) transactions and undo necessary operations (i.e., all writes that are not overwritten by a redo)

Evaluated undo's and redo's are also **logged**, so that a crash during update **does not** cause a need to **re-evaluate**

ARIES recovery protocol: Example

Lsn	Last_lsn	Tran_id	Type	Block_id	Other_information
1	0	T_1	write	C	...
2	0	T_2	write	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	write	A	...
7	2	T_2	write	C	...
8	7	T_2	commit		...

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	2	in progress

DIRTY BUFFER TABLE

Block_id	Lsn
C	1

Let the log above be in on the disk after a crash. ARIES steps:

1. **Analysis step**: identifies the earliest log entry: Lsn 1
2. **Redo step**: redo the writes with Lsn 1, 6, 7 (no 2)
3. **Undo step** (only T_3): undo update with Lsn 6

(Lsn 6 is redone and undone;

think of advantages and disadvantages of such strategy)

6. Other considerations

Recovery in multidatabase cases

Often, a transaction accesses several databases, in which a two-level recovery mechanism can be used:

- managed by (global) coordinator
- consists of two phases: in phase 1, **coordinator** prepares for commit by sending a message to all databases, which respond 'ready-to-commit' or 'cannot-commit';
if all successful, in phase 2 **coordinator** issues commit signal to all participating databases
- always possible to recover to a state where the transaction is committed or aborted
- Failure during Phase 1 requires rollback
- Failure during Phase 2 means
 successful transaction can recover and commit

Database backup:

- entire database and log periodically copied onto inexpensive storage medium
- latest backup copy can be reloaded to disk in case of catastrophic failure
- often moved to physically separate locations
(e.g., subterranean storage vaults)
- system log is usually smaller than the database
so a second backup may be the log after a first backup
- may benefit from more frequent backups
(e.g., to magnetic tapes)

What have we learned?

Recovery protocols

that ensure **atomicity** and **durability**:

- **shadowing-based**

easy but essentially no cache, so slow

- **deferred-update**

easier than immediate-update but slower to recover

- **immediate-update**

general

- **ARIES**

immediate-update with improvements

Rest of the course [Part 3](#) (and a little more):

- No-SQL databases overview

IN3020&4020 – Database Systems (2024)

(Between Parts 2 and 3)

Lecture 24: Database security

16 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓
- Part 1. Query processing in relational databases ✓
- Part 2. Transaction management in relational databases ✓
 - **Database security**
- Part 3. NoSQL DBMS
 - Advanced DBMS architecture (IN4020 only)

Lecture schedule

#	Date	Part	Topic
...
24	16.04 (Tue)	(no part)	Database security
25	22.04 (Mon)	Part 3: NoSQL	Overview
26	23.04 (Tue)	Part 3: NoSQL	Knowledge Graphs
27	29.05 (Mon)	Part 3: NoSQL	RDF and SPARQL
28	30.04 (Tue)	Part 3: NoSQL	Neo4j and Cypher
29	06.05 (Mon)	for IN4020	Advanced DBMSs (digital, intro for IN5040)
30	07.05 (Tue)	Wrap-up	Summary
31	13.05 (Mon)	2024 exam discussion	Summary

Exam: Thursday, 30 May (see next slide)

Exam formal details:

- **Time:** 30 May 2024, 09:00 **Duration:** 4 hours
- **Place:** Silurveien **System:** Inspera
- **Withdrawal deadline:** 1 May

Exam rules and contents:

- no **collaboration** (standard plagiarism rules)
- the slides will be available
 - the content in the spirit of the
mandatories, * questions in weekly exercises & 2024 exam
- if you are comfortable with **mandatories**, you should be fine with the exam
- **no one** is expected to do everything correct
(and it is not required for A)

Materials to read about database security

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition):
Section 30 (the only in Part 12)
(more than my presentation)
2. *Database Systems: the Complete Book* by H. Garcia-Molina, J. Ullman & J. Widom (2nd edition): Parts of Section 10
3. IN2090 – Databaser og datamodellering (Leif Harald Karlsen):
Lecture 12 (basics)
<https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/>
4. etc.

1. Introduction and main concepts
2. Discretionary access control (DAC)
3. Mandatory access control (MAC)
4. Other control measures

1. Introduction and main concepts

Database security is a broad area with many issues:

- Legal, ethical, and policy issues
(e.g., private information access rights, corporate policies)
- System-related issues
(e.g., system levels on which security should be enforced)
- Security categorisation and organisation
(classes of users vs. classes of information)

Threats to databases:

- Loss of integrity: Improper modification of information
- Loss of availability: Legitimate user cannot access data objects
- Loss of confidentiality: Unauthorised disclosure of confidential information

Database Security: Not an isolated concern

Applications often contain much **more** than just the database:
the database works as part of a **network of services**
(other applications, programs, Web servers, etc.)

- Database security can therefore **not only** focus on the database
- Security holes can occur in **all joints** (frontend, backend, network, etc.)
- Safety is therefore always a **comprehensive task**:
must be ensured for **each component**
and the **interaction** between them
- Must be clear on what **assumptions**
about other components each part of the system makes

DB-related threat example: SQL injection

SQL injection:

- Attacker **changes** SQL statement to their advantage
(e.g., commonly, adds **conditions** to the **WHERE** clause)
- May lead to **unauthorised** data manipulation
or execution of **system-level** commands

Example:

```
SELECT * FROM Users WHERE  
    username = 'jake' AND password = 'jakespasswd'
```

can be **changed** to

```
SELECT * FROM Users WHERE  
    username = 'jake' AND (password = 'jakespasswd' OR 'x' = 'x')
```

with obvious **consequences**

Database multi-user security mechanisms

DBMS must provide techniques to enable certain **users** or **user groups** to access selected **portions** of a database

There are several **types** of security **mechanisms**; most fundamental:

- **Discretionary** security mechanisms
 - Used to **grant privileges** to users to access certain pieces of data
 - Commonly used in **practice**
- **Mandatory** security mechanisms
 - Classify data pieces** and **users** into various security **classes**
 - Give access according to a corresponding **security policy**
 - Used (and extended) in **professional DBMSs** (e.g., **Oracle**)

We will briefly consider such mechanisms for **access control** below

What is access? Control measures for data security

Access control (narrow sense, our main focus below):

- handled by **user accounts** (with **passwords**) and data labelling to ensure the **security mechanism**

Inference control (briefly at the end):

- ensure that information about **individuals** cannot be **derived** key in **statistical databases**

Flow control (briefly at the end):

- prevents information from **flowing** from authorised to unauthorised users disallow **covert channels**

Data encryption (briefly at the end):

- used to protect sensitive data during **transmission** symmetric/asymmetric encryption, digital signatures and certificates

Database administrator (DBA):

- central **authority** for **administering** database system
- **superuser** or system account
- **central role** for ensuring security (especially access control)

DBA-privileged commands for **access control**:

- **account creation**
- **privilege granting** and **revocation** (discretionary authorisation)
- **security level assignment** (mandatory authorisation)

Users: Security-related system tables and audit

User must **log in** using assigned **username** and **password**

DBMS has **encrypted tables** for

- **users**, including accounts and passwords
- a write-only log of **login sessions** (user ID, beginning, end, etc)
 - a logic session may include several transactions
 - may re-use and extend the recovery **system log**

If **tampering** is suspected, **database audit** is performed:
reviewing **log** to examine all accesses and operations
to determine the **possible culprit**

Sensitivity of data:

measure of the **importance** assigned to the data by its owner denoting its **need for protection**

Different parts of data in database may have **different sensitivity**

Factors that can cause **sensitivity status** of a piece of data:

- **inherently sensitive** (e.g., a patient's diagnosis)
- from a **sensitive source** (e.g., from a secret informer)
- **declared sensitive** (e.g., by the data owner)
- has a **sensitive part** (e.g., salary attribute)
- sensitive in **relation to previously disclosed data**
(i.e., disclose sensitive information
in conjunction with previously given data)

Data: Security vs. precision

Security:

ensures that sensitive data is kept **safe** from **corruption** and **non-authorised access**

Precision (or Availability):

To make available as much nonsensitive data as possible

(In both cases, sensitivity is **relative** to a specific user)

The **ideal combination** is to maintain **perfect security** with **maximum precision**
contradictory in practice, so the best **trade-off** should be found

Security as a part of privacy

Security:

issues related to **access to resources**

Privacy goes **beyond**:

issues related to appropriate **use of information**—
that is, the ability of individuals to control the terms
under which their **personal information** is acquired and used;
for example

- preventing **storage** without a need
- ensuring **appropriate use**

General Data Protection Regulation (GDPR):

an EU (and EEA) **data protection** and **privacy law**
(should always be considered when considering privacy issues)

2. Discretionary access control

Discretionary access control: Idea

Part of SQL standard and commonly implemented
(sometimes partially)

Discretionary access control (DAC):

- based on granting and revoking privileges to users (or groups of users)
- SQL has statements GRANT and REVOKE for granting and revoking

(Revoking may be necessary when temporal access is needed)

There are two levels of privileges: account level and relation level

Account-level privileges—

that is, privileges applying to the **whole database**:

- privileges for
 - data definition** (CREATE/DROP/ALTER TABLE/VIEW)
 - data modification** (INSERT/UPDATE/REMOVE)
 - data access** (SELECT)
- **granted/revoked** by the DBA

Examples:

```
GRANT CREATETAB TO UserName
```

```
REVOKE CREATETAB FROM UserName
```

Relation- (table-) level privileges

Relation-level privileges—

that is, privileges applying to **specific tables** and **views**:

- privileges for
 - data modification** (INSERT/UPDATE/REMOVE)
 - data access** (SELECT)
 - constraints** (FOREIGN KEY)
- can be seen as a **access matrix** (Users \times Tables)
- (originally) **granted/revoked** by the **owner** (usually the **creator**)

Example:

```
GRANT INSERT, DELETE
      ON Employee, Department TO UserName
```

Propagation of relation-level privileges

Relation-level privileges can be propagated to other users by `WITH GRANT OPTION` modifier

Example:

```
GRANT SELECT ON Employee, Department TO UserName
    WITH GRANT OPTION
```

Revoking is automatically cascading

Example:

```
REVOKE SELECT ON Employee FROM UserName
```

automatically **recursively** revokes all privileges given by `UserName`

Should be **careful** with **multiple** grantees and should prevent **cycles**

Some DBMSs allow for **horizontal** (i.e., number of users) and **vertical** (i.e., depth) **limits** on propagations

Fine-grained privileges via views

Views are a DAC mechanism by themselves:
can give fine-grained access to certain attributes and/or tuples

Examples:

Consider owner UserA of relation R and other user UserB

- UserA can create view V1 of R
that includes only attributes UserA wants UserB to access
and grants SELECT privilege on V1 to UserB
- UserA can also create view V2 of R
that selects only tuples UserA wants UserB to access
and grants SELECT privilege on V2 to UserB

3. Mandatory access control

Mandatory access control: Idea

Mandatory access control (MAC) idea:

- **classify** subjects (i.e., users) and **objects** (i.e., data elements such as attribute values and tuples) to security **classes**
- classes are (possibly partially) **ordered** (i.e., **multilevel security**)
- compose the **classes** of involved subjects and objects using the order when deciding to **give access**

Typical (simple) security classes

- Top secret (TS)
- Secret (S)
- Confidential (C)
- Unclassified (U)

with (total) **order** $TS \geq S \geq C \geq U$

Bell-LaPadula access control model

Two **rules** are enforced

(general rules, may be applied to any type of data)

– **Simple security** property:

subject (e.g., user) S can **read object** (e.g., value) O

only if $class(S) \geq class(O)$

that is, **no one** can read data with **higher security**

– **Star** property:

subject (e.g., user) S can **write object** (e.g., value) O

only if $class(S) \leq class(O)$

that is, no one can write data with lower security

(prevents **information flow** from higher to lower classifications)

MAC model for relational databases

Mandatory access control (MAC) in DBMSs:

- less commonly adopted due to complexity (mostly professional)
- objects are attribute values in tuples and tuples themselves
- classification of a tuple is
the highest classification of its attribute values

Example: $(TS \geq S \geq C \geq U)$

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

(We omit the commands for assigning classes,
they are not in the SQL standard anyway)

Reading in MAC model for relational databases

Filtering is used for **reading** (i.e., **SELECT** queries):

- **replacing** by **NULLs** attribute values with higher classification than user's
- **removing** whole tuples when **nothing** left

Example:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

Query **SELECT * FROM Employee** issued by user of **class C** gives

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

Reading in MAC model for relational databases

Filtering is used for **reading** (i.e., **SELECT** queries):

- **replacing** by **NULLs** attribute values with higher classification than user's
- **removing** whole tuples when **nothing** left

Example:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

Query **SELECT * FROM Employee** issued by user of **class U** gives

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

Writing in MAC model for Relational Databases

Poly-instantiation can be used for **writing** (e.g., UPDATE queries):

- updates on the **lower level** is **not allowed** (star property)
- updates on the **same level** is **as usual**
- updates on the **higher level** creates **several copies**
(need to be **careful**, details are omitted)

Example:

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

Query UPDATE Employee SET Job_performance = 'Excellent'
WHERE Name = 'Smith' issued by user of **class C** gives

EMPLOYEE

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

Discretionary access control (DAC):

- more **fine-grained**
- more **vulnerable**
- **no** flow control

Mandatory access control (MAC):

- more **rigid**
- higher degree of **protection**
- **ensures** flow control

Professional DBMSs (e.g., **Oracle**) implement more **advanced** models, often **generalising** both DAC and MAC:

- **Role-based** access control
- **Label-based** (e.g., for tuples) access control

4. Other control measures

Statistical database

is a **usual** database with special use:

intuitively, (usual) users are

- assumed to retrieve only **statistical information**
- forbidden to retrieve **individual data**

Essentially, implemented so that only **statistical** (i.e., **aggregate**) queries are allowed; for **example**

```
Q1: SELECT COUNT(*) FROM PERSON  
WHERE <condition>;
```

```
Q2: SELECT AVG (Income) FROM PERSON  
WHERE <condition>;
```

Forbidding non-aggregate queries is **not enough**

Example:

```
Q1: SELECT COUNT(*) FROM PERSON  
WHERE <condition>;
```

```
Q2: SELECT AVG (Income) FROM PERSON  
WHERE <condition>;
```

If count is 1, the average is the **exact number**

Can be resolved by **several methods**, for example:

- fixing a **minimum threshold** on number of tuples
- introducing **noise**
- prohibit **sequences** of queries for the same population
(e.g., table)

Flow control:

- regulates the **flow** (i.e., **distribution**) of information among accessible objects
- ensures that information contained in some objects does not flow explicitly or implicitly into **less protected** objects

Flow policy:

- specifies **channels** along which information is allowed to move
- **covert channels** (improperly) allows to pass from higher to **lower** classification

We have seen an **example**:

MAC

Encryption of sensitive data

Encryption converts data into cyphertext

- performed by applying an **encryption algorithm** to data using a pre-specified **encryption key**
- resulting data can only be **decrypted** using a **decryption key** to recover original data

Encryption protocols and algorithms

- a big separate **research field**
- **classic approach**: **symmetric** (secret key) protocols
requires the same secret key for encryption and decryption
- **modern approach**: **asymmetric** (public key) protocols
RSA scheme (Rivest, Shamir, and Adleman, 1978):
encryption with public key, decryption with private key
deep mathematics (number theory, one-side functions)

What have we learned?

Brief intro to **database security**

Next time: **Part 3 (NoSQL)**

IN3020&4020 – Database Systems (2024)

Part 3: NoSQL

Lecture 25: Overview

22 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓
- Part 1. Query processing in relational databases ✓
- Part 2. Transaction management in relational databases ✓
 - Database security ✓
- Part 3. **NoSQL DBMS**
 - Advanced DBMS architecture (IN4020 only)

Lecture schedule

#	Date	Part	Topic
...
24	16.04 (Tue)	(no part)	Database security
25	22.04 (Mon)	Part 3: NoSQL	Overview
26	23.04 (Tue)	Part 3: NoSQL	Knowledge Graphs
27	29.05 (Mon)	Part 3: NoSQL	RDF and SPARQL
28	30.04 (Tue)	Part 3: NoSQL	Neo4j and Cypher
29	06.05 (Mon)	for IN4020	Advanced DBMSs (digital, intro for IN5040)
30	07.05 (Tue)	Wrap-up	Summary
31	13.05 (Mon)	2024 exam discussion	Summary

Exam: Thursday, 30 May

Materials to read about NoSQL

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition):
Section 24 (and 25) (close to my presentation)
2. (English) Wikipedia
3. Systems' tutorials (MondoDB, SPARQL, etc.)
4. etc.

1. Introduction and classification
2. Document-based systems
3. Key-valued stores
4. Column-based systems
5. Graph-based (a quick look)

1. Introduction and classification

New main-line **types of applications** (data science, social media) impose **new requirements** upon the underlying systems (including DBMS)

NoSQL (**Not only** SQL) serves these needs

Most **NoSQL** systems are **distributed** over several nodes

Main **challenges** (and motivation for NoSQL):

- **semi-structured** data (no or little schema)
- **high performance**
- data **availability** and **replication** (several unreliable nodes)
- **scalability** (as opposed to immediate consistency)
- **sharding** and range **partitioning**

NoSQL systems focus on **Big Data**:

- data sets too **large** or **complex** for traditional data-processing
- **challenges**: capturing data, data storage, data analysis, search, sharing, transfer, visualisation, querying, etc.
- **V-characteristics**:
 - **Volume** (currently, the size 10^{15} B– 10^{18} B),
 - **Velocity** (speed of creation, ingestion, and processing)
 - **Variety** (various types)
 - (Veracity, Value, Variability, etc.)

Some typical **applications** that use NoSQL:

- **Social media** (Facebook, etc.)
- **Web links** (Google search)
- **Marketing and sales** (Amazon, etc.)
- **Interactive maps** (Google maps, etc.)
- **Email** (Gmail, etc.)
- **Ontologies and Knowledge Graphs** (Equinor, Bosch, etc.)

NoSQL characteristics related to **distributed** database systems:

- **Scalability** (horizontal, i.e., adding more distributed data nodes)
- **Availability** via **replication** and **eventual consistency**
(c.f., **serialisable consistency**)
- **Replication models**
 - **Master-slave** (write to one master, propagate to slaves)
 - **Master-master** (different writes, reconciliation)
- **Sharding** of files (one table distributed over several files)
- High performance **data access** (e.g., via indexes)

NoSQL: data model and query language characteristics

NoSQL characteristics related to **data models** and **query languages**:

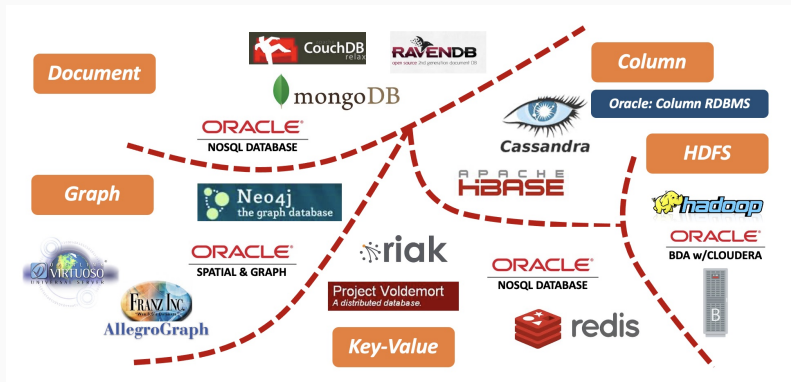
- Schema **not required**: data may be
 - **semi-structured**
for example, **JSON** and **XML** data formats
 - **self-describing** ('schema' as a part of data)
for example, **RDF/S** knowledge graphs
- **Less powerful query** languages:
 - usually **search** rather than **querying** (e.g., no join)
 - **(S)CRUD**: (search), create, read, update, delete
- **Versioning**
 - e.g., timestamps

Categories of NoSQL systems

- Document-based systems: MongoDB, CouchDB, etc.
- Key-value stores: DynamoDB, Voldemort, etc.
- Column-based (or wide-column) systems: HBase, Cassandra, etc.
- Graph-based systems: RDF, Neo4j, etc.
- Hybrid systems: HADOOP, etc.
- (Object databases, XML databases)

NoSQL: Landscape

Some **examples** in the landscape:



CAP theorem

Levels of **consistency** among **replicated** (distributed) data items

- the **strongest** form: **serialisability**
 - high **overhead**: can reduce read/write performance
- (on this slide, 'consistency' is not the C in ACID)

CAP theorem (more like a **principle** here, since only informally):

The three desired properties
of distributed systems with replicated data:

- **consistency**: every read gets most recent write
- **availability**: every read and write receives a response
- **partition tolerance**: works after a (disconnected) split

are **not possible** to guarantee simultaneously
(in distributed system with data replication)

CAP theorem:

consistency, availability, and partition tolerance
are **not possible** to guarantee simultaneously

NoSQL system designers can choose two of three to guarantee

- **availability** and **partition tolerance** more important
- **weaker consistency** levels is often acceptable
- **eventual consistency** is often adopted

Document-based NoSQL systems

Document-based systems: Idea

Document stores:

Collections of similar **documents**

Individual documents may be **complex objects**:

- documents may have internal **structure** and be **self-describing**
- can have **different** data elements (attributes)

Documents can be specified in various formats

- **XML (Extensible Markup Language)**
file format for storing, transmitting, and reconstructing arbitrary data
tree-like semi-structured data
- **JSON (JavaScript Object Notation)**
open standard human-readable file format and data interchange format
attribute-value pairs and arrays

MongoDB

- a popular **document-based** approach
- documents stored in **binary JSON (BSON)** format

Individual documents stored in **collections**

- collection creation **example**:

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

- **first** parameter specifies the collection name
- **collection options** may include limits
on size and number of documents
- (each document in collection has **unique ID**)

MongoDB data model: CRUD

Collection of documents **does not** have a **schema**

- structure of documents are chosen based on how documents will be accessed

Document **creation** using insert operation (documents in BSON)

```
_ db.<collection_name>.insert(<document(s)>)
```

Document **reading** using find operation with a Boolean condition (and indexing)

```
_ db.<collection_name>.find(<condition>)
```

Document **deletion** using remove operation

```
_ db.<collection_name>.remove(<condition>)
```


Two **collections** (projects and workers) with interreferences:

```
Collection db.project: {
  _id:      "P1",
  Pname:    "ProductX",
  Plocation: "Bellaire"
}

Collection db.worker: [{
  _id:      "W1",
  Ename:    "John Smith",
  ProjectId: "P1",
  Hours:    32.5
},
{
  _id:      "W2",
  Ename:    "Joyce English",
  ProjectId: "P1",
  Hours:    20.0
}]
```

Note: Different workers can have different keys

Can be achieved via **inserts**:

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
  { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
    Hours: 20.0 } ] )
```

Concurrency in MongoDB:

- Two-phase commit for atomicity and consistency

Replication in MongoDB:

- Replica sets for **multiple copies** on different nodes
- Variation of **master-slave** approach:
 - primary copy, secondary copy, and arbiter
 - (arbiter participates in elections of a new primary if needed)
- All write operations applied to the primary copy and propagated to the secondaries
- User can choose read preference
 - (read requests can be processed at any replica)

Sharding in MongoDB:

- **horizontal partitioning** divides a collection into disjoint partitions (**shards**) of documents
- shards stored on **different nodes** to achieve load balancing
- **partitioning field** (shard key) exists in every document in the collection and be indexed
- **range** or **hash** partitioning is used

Key-Value NoSQL stores

Key-Value stores: Overview

Key-value stores:

focus on **high performance**, **availability**, and **scalability**
(as opposed to complex querying and search)

- **Key**: **unique identifier** associated with a data item
used for **fast retrieval**
- **Value**: the **data item** itself
can be **string** or **array of bytes**

No built-in **query language**

fast **retrieval by key** is the main goal and characteristic
application is responsible for searching, querying, etc.

Several similar systems, including

- **DynamoDB** (Amazon, proprietary)
- **Voldemort** (open-source)

DynamoDB:

part of Amazon Web Services and SDK platform (**proprietary**)

Uses its own **terminology**:

- **Table**: collection of items (with no schema)
- **Item**: set of attribute-value pairs
- **Attribute**: simple or complex unique identifier (primary key)
- **Value**: arbitrary data values

Implements **quick access** via hashing and indexing

Voldemort

open-source key-value system similar to **DynamoDB**

focus: horizontal scalability, replication, sharding

Voldemort **features:**

- Simple **basic** CRUD operations (get, put, and delete by key)
- High-level formatted **data values** (input as **JSON**)
- **Hashing** for distributing key-value pairs across nodes (i.e., sharding)
 - also used for replication (like master-slave, but more advanced)
- **Consistency and versioning:**
 - **concurrent** writes allowed
 - each read may return **several versions**

Other Key-Value stores

Redis key-value cache and store

- caches data in main memory to improve performance
- master-slave replication and high availability
- persistence by backing up cache to disk

Apache Cassandra

- offers features from several NoSQL categories
- used by Facebook and others

Oracle NoSQL database

- high-level of integration into other Oracle products

etc.

Column-based (wide-column)
NoSQL systems

Column-based systems: Overview

Column-based systems:

for storing **large amount of data**

similar to **key-based**, but with multi-dimensional keys

(e.g., table name, row key, column info and time-stamp)

BigTable: Google's distributed storage system for big data

- Used in **Gmail**
- Uses **Google File System** for data storage and distribution
- (omitted in our presentation)

Apache Hbase: a similar, open-source system

- Uses **Hadoop Distributed File System (HDFS)** for **data storage**
- Can also use Amazon's **Simple Storage System (S3)**

Hbase Data Model

Data organisation **concepts**:

- **Namespace**: collection of tables
- **Named table**: set of self-describing **rows** with **row keys** (ordered lexicographically)
- **Named column families** (associated with tables)
 - created on creation of table and cannot be changed
 - used for grouping together related columns (attributes)
- each column family can be associated with **column qualifiers**
 - make the model self-describing
 - not specified on creation, can be different in rows
- a column is a combination **ColumnFamily:ColumnQualifier**
- a **timestamp** identifies a version of a data item

Each column family stored in its own file(s)

using the **HDFS** file system

Hbase Provides only **low-level CRUD** operations
(Applications can implement more complex operations)

Create operation:

Creates a **new table** (with associated **column families**)

Put operation:

Inserts **new data** (or **new versions** of existing data)

Get operation:

Retrieves data associated with a **single row**
(most recent by default, but older versions also possible)

Scan operation:

Retrieves **all rows** from a table

Delete operation:

Make an item 'invisible'

HBase: Example

(a) creating a table:

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) Inserting some row data in the EMPLOYEE table:

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'  
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'  
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'  
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'  
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'  
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'  
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'  
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'  
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'  
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'  
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'  
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'  
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'  
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'  
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'  
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) Some Hbase basic CRUD operations:

Creating a table: create <tablename>, <column family>, <column family>, ...

Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>

Reading Data (all data in a table): scan <tablename>

Retrieve Data (one item): get <tablename>,<rowid>

HBase: Storage

Each Hbase table **divided** into several **regions**

- each region holds a **range** of the row keys in the table
- row keys must be **lexicographically ordered**
- each region has **several stores**
(column families are assigned to stores)

Regions assigned to **region servers** for storage

- master server responsible for monitoring the region servers

Hbase uses

- **Apache Zookeeper** (open source system for services related to data naming, distribution, and synchronisation)
- **HDFS (Hadoop Distributed File System)** for file services

Graph databases: A quick look

Graph databases idea:

- Data represented as a (labelled) **graph**
(collection of **vertices** (nodes) and **edges**)
- **Labels** store data associated with nodes and edges

Two main **types**:

- **RDF Graphs**
Older, W3C standardised
has a standardised query language **SPARQL**
- **Property graphs**
Newer, ISO standardised (very recently!)
has several query languages:
PGQL (Oracle), **Cypher** (Neo4j), **Tinkerpop** (Apache), etc.

Next time

What have we learned?

Brief intro to NoSQL

Next time: Graph databases (in detail)

IN3020&4020 – Database Systems (2024)

Part 3: NoSQL

Lectures 26–28: Graph Databases (and Knowledge Graphs)

23, 29, 30 April

Egor V. Kostylev

IFI, University of Oslo

egork@ifi.uio.no



Big Picture (reminder)

Syllabus of the course:

- (Extended) Intro and SQL recap ✓
- Part 1. Query processing in relational databases ✓
- Part 2. Transaction management in relational databases ✓
 - Database security ✓
- Part 3. **NoSQL DBMS**
 - Advanced DBMS architecture (IN4020 only)

Materials to read about NoSQL

1. *Fundamentals of Database Systems* by R. Elmasri and S.B. Navathe (7th edition):
Section 24 (few pages only)
2. (English) Wikipedia
3. Systems' tutorials
 - RDF and SPARQL are W3C standards
e.g., <https://www.w3.org/TR/rdf11-concepts/>
 - Neo4j and Cypher also gave good documentation
4. etc.

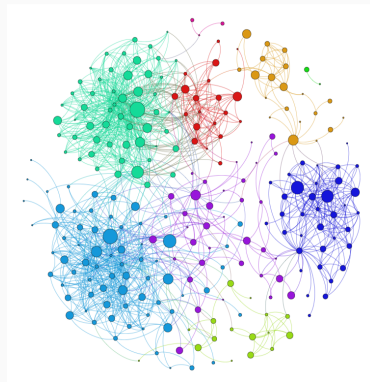
1. Overview of knowledge graphs
2. Semantic knowledge graphs
 - 2.1. RDF
 - 2.2. Implicit Knowledge: RDFS and OWL
 - 2.3. Query language: SPARQL
3. Property graphs
 - 3.1. Neo4j
 - 3.2. Cypher

1. Overview of knowledge graphs (a.k.a. graph databases)

Why graphs?

Graphs:

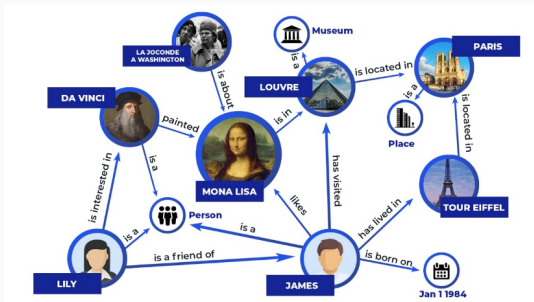
- **universal** mechanism for describing complex data
- shared **terminology** between disciplines
- intuitive (!)
- many data are **naturally** graphs
 - social media
 - economic networks
 - the Web
 - molecules
 - networks of neurons
 - etc.



Knowledge graphs

Knowledge graphs (KGs), or Graph Databases

- annotated, usually directed, often multi- **graphs**
- studied in Knowledge Representation and Reasoning **AI** branch
- capture **data** (facts) and **semantics** (metadata, schema, rules)
- **nodes** connected by **relationships**



Gartner Top 10 **Data and Analytics** Technology Trends for 2019

Source: Gartner (February 2019)

- | | |
|----------------------------------|---------------------------------|
| 1. Augmented Analytics | 6. Data Fabric |
| 2. Augmented Data Management | 7. Explainable AI |
| 3. NLP/ Conversational Analytics | 8. Blockchain in Data Analytics |
| 4. Graph analytics | 9. Continuous Intelligence |
| 5. Commercial AI and ML | 10. Persistent Memory Servers |

Still there in recent years (in some form)

Graphs:

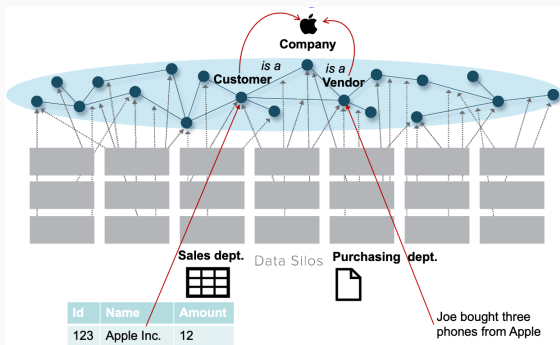
- Underlie **Graph Databases** as a **NoSQL** approach
- Used in the new generation of **Enterprise** and **Web applications**
- Exploited in **representational** and **structure-aware learning**

KGs in enterprise example

One of the main use is **data integration**:

- departments keep data in own databases in **different formats**
- they specify the **mappings** (i.e., 'views') to a **common KG**
- the KG can be **accessed** (e.g., queried) **without** knowing the individual schemas

Example:



KGs in the Web (and Enterprise) example

Google Knowledge Graph:

- *'A huge knowledge graph of interconnected entities and their attributes'*

Amit Singhal, Senior Vice President at Google




- *'A knowledge base used by Google to enhance its search engine's results with semantic-search information gathered from a wide variety of sources'*

http://en.wikipedia.org/wiki/Knowledge_Graph

- used in many Google applications, including search and voice assistant
- emerged from open Freebase KG, is based on information derived from many sources (e.g., Wikidata, Wikipedia)
- contains over 70 billion facts (i.e., edges)

Google KG in Search: Example 1

Entity search and summarisation



[All](#) [Images](#) [Maps](#) [News](#) [Videos](#) [More](#) [Settings](#) [Tools](#)

About 400,000,000 results (0.84 seconds)

Oslo - Wikipedia

<https://en.wikipedia.org/wiki/Oslo>

Oslo is the capital and most populous city of Norway. It constitutes both a county and a municipality. Founded in the year 1040 as Årsto, and established as a ...

Area code(s): (+47) 00 **Elevation:** 23 m (75 ft)
Country: Norway **Established:** 1048





History of Oslo's name · Oslo Metro · Oslo City · University of Oslo



Oslo, Norway - Official travel guide

<https://www.visitoslo.com>

Official travel guide for Oslo with updated info on hotels and accommodation, map, tourist information, congress, attractions, activities, concerts.

Top things to do in Oslo

 <p>The Vigeland Park Park & museum of Vigeland's sculpture</p>	 <p>Viking Ship Museum Museum with 3 9th-century Viking ships</p>	 <p>Akershus Fortress Waterside fortress & former prison</p>	 <p>The Royal Palace Royal residence open to public in...</p>
--	--	---	--



Oslo

Capital of Norway

Oslo, the capital of Norway, sits on the country's southern coast at the head of the Oslofjord. It's known for its green spaces and museums. Many of these are on the Bygdøy Peninsula, including the waterside Norwegian Maritime Museum and the Viking Ship Museum, with Viking ships from the 9th century. The Holmenkollbakken is a ski-jumping hill with panoramic views of the fjord. It also has a ski museum.

Weather: 0°C, Wind NE at 4 m/s, 61% Humidity
Local time: Tuesday 11:59
District: Østlandet
Population: 673,469 (2018) Eurostat
Postal code: 0001 – 1299
Mayor: Marianne Borgen (SV)

Google KG in Search: Example 2










Discovering related entities

Google

oslo colleges and universities

Q All Maps Images News Videos More Settings Tools

Oslo / Colleges and Universities


 University of Oslo	 OsloMet – storbyunivers...	 BI Norwegian Business School - Osl...	 Oslo University College	 Oslo School of Architecture and Design	 Norwegian Academy of Music	 Oslo National Academy of the Arts	 Norwegian Police University C...	 Westerdals Oslo ACT
---	---	--	--	---	---	---	---	--

List of universities in Norway - Wikipedia
https://en.wikipedia.org/wiki/List_of_universities_in_Norway ▼
This list of **universities** in Norway presents the country's **universities**, giving their locations, abbreviated titles (in **Norwegian**), and years of establishment. Most **universities** in Norway are public. Most of the **university colleges** were created in 1994, following the **university ... Norwegian Police University College** - Oslo, Bodo, Public, University college ...

People also ask

Do universities in Norway teach in English? ▼

Oslo
Capital of Norway



Google KG in Search: Example 3

Answering factual questions

Google


when was oslo founded?

All Images Maps News Shopping More Settings Tools

About 12,600,000 results (0.67 seconds)


Oslo / Established

1049



According to the Norse sagas, **Oslo** was **founded** around 1049 by Harald Hardrada. Recent archaeological research however has uncovered Christian burials which can be dated to prior to AD 1000, evidence of a preceding urban settlement.

[Oslo - Wikipedia](https://en.wikipedia.org/wiki/Oslo)
<https://en.wikipedia.org/wiki/Oslo>



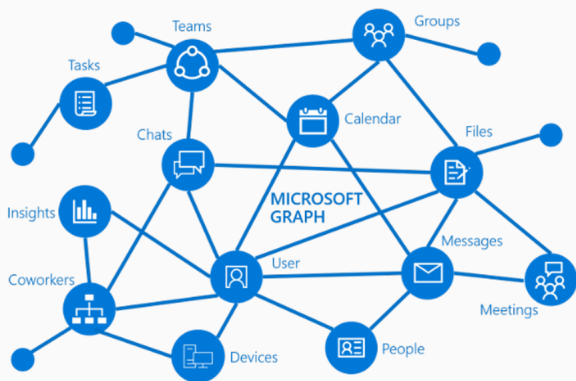
Oslo
Capital of Norway

Weather: 0°C, Wind NE at 4 m/s, 60% Humidity
Local time: Tuesday 12:03

Plan a trip
Oslo travel guide

Other enterprise KGs

Microsoft graph:

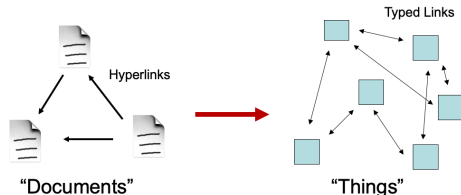


Siemens, LinkedIn, Airbnb, eBay, Apple, etc.

Linked data principles

Enterprise KGs are usually a closed implementation of **Web of Data** (Semantic Web) and **Linked Data** principles

Web of Documents → **Web of Data / Linked Data**



Linked Data principles (by **Tim Berners-Lee**):

- all conceptual things should have a **name** (e.g., starting with HTTP)
- looking up a name should return **useful data** about the thing in question in a **standard format**
- anything a thing **relates** (i.e., links) to should also be given a name (and be accessible)

Linked Data:

- method for **publishing data** on the Web
- **self-describing** semi-structured data:
 entities and **relations** between them
- can be **accessed** using **semantic queries**

W3C (WWW consortium) developed **standards** for Linked Data
(<http://www.w3.org/standards/semanticweb/data>):

- data format: **RDF**
- knowledge: **RDFS** and **OWL**
- query language: **SPARQL**



RDF graphs: Idea

RDF graph: a set of triples

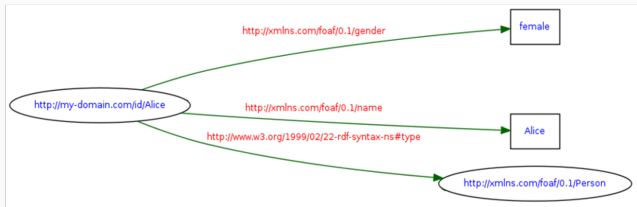
subject predicate object

of IRIs or literals (see below), can be seen as



Example:

```
http://my-domain.com/id/Alice http://xmlns.com/foaf/0.1/gender "female" .  
http://my-domain.com/id/Alice http://xmlns.com/foaf/0.1/name "Alice" .  
http://my-domain.com/id/Alice  
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type  
        http://xmlns.com/foaf/0.1/Person
```

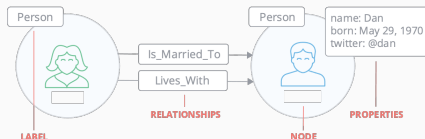


Property Graphs: Idea

Property graphs:

- more **modern** approach
- no IRI's, more **structure**
- **no** standard yet (!)
- **implemented** in several systems
(e.g., **Neo4j** with its own query language **Cypher**)

Example (with terminology):



Some **constraints**, such as

- each relationship has **exactly** one type
- each node can have **none or several** labels and/or properties

What can we **do** with graphs?

- **querying** using SQL-like languages (SPARQL, Cypher)
- more complex **graph algorithms**
- **ML** over graph

Next we briefly look at the **latter two**,
leaving more detailed discussion of query languages for **later**

Path & subgraph finding:

find the **shortest path(s)** or other **'best' subgraph(s)**

(e.g., single source or all-pairs shortest path, Minimum-weight spanning tree)

Centralities:

determine the **importance** of distinct nodes in a network

(e.g., Closeness Centrality, Betweenness Centrality, Google's PageRank)

Community detection:

evaluate how a group is **clustered** or **partitioned**,

as well as its tendency to **strengthen** or **break apart**

(e.g., Louvain, label propagation, triangle count)

Example: Centralities

Shortest path (simplest version):

- a shortest path between 2 nodes in undirected unweighted graph

Closeness Centrality: a number for a node

- the average shortest path length between the node and all nodes

Betweenness Centrality: a number for a node

- the sum of proportions of shortest paths (for all pairs of nodes in the graph different from the node) that pass through the node:

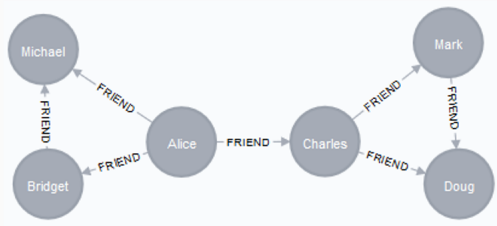
$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} the number of shortest paths from s to t

σ_{st} is the number of those paths that pass through v

Example: Louvain method for community detection

- Used for detecting **communities in networks**
- Evaluates how much more **densely connected** the nodes within a community are, compared to how connected they would be in a random network



Name	Community
Alice	0
Bridget	0
Michael	0
Charles	1
Doug	1
Mark	1

Machine Learning on Graphs

There are many ML tasks on graphs:

- most natural:
node classification (predict the type of a given node)
- many others:
link prediction, network similarity, etc.
- classic approach:
embed the input (graph and nodes) into a vector space,
apply a standard NN
- structure-aware approach:
use Graph Neural Networks (GNNs)
come to me if interested

Machine Learning on Graphs: Examples

Link prediction

(1-1, 1-n, n-m)



Attribute prediction

age=63

age=65

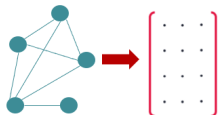


Subgraph prediction

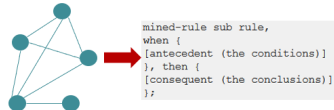
age=64



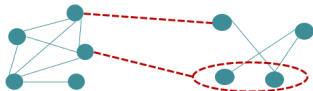
KG embeddings



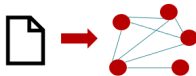
Rule mining



Graph matching



Automated KG creation from text



Optimal path finding on KGs

Query answering on KGs

....

Graph databases (knowledge graphs) idea:

- Data represented as a (labelled) **graph**
- **Labels** store data associated with nodes and edges

Two main **types**:

- **RDF Graphs**
 - Older, W3C standardised
 - has a standardised query language **SPARQL**
- **Property graphs**
 - Newer, not yet standardised
 - has several query languages **Cypher** (by **Neo4j**), etc.

An **emerging technology**

- becoming **popular** in industry and academic applications
- **developed** by many large and small companies
- KGs and **ML** complement each other

2. Semantic KGs (RDF, RDFS/OWL, SPARQL)

2.1. RDF Graphs

RDF: a language that enable us to make statements about **resources**

Example statement: “*John is father of Bill*”

RDF graph is a set of triples (here in Turtle-like syntax):

subject₁ predicate₁ object₁ . subject₂ predicate₂ object₂

- subject: **resource** (IRI) or **blank node**
- predicate: **resource**
- object: **resource**, **literal** or **blank node**

Triple *subject predicate object* is often written as
(*subject, predicate, object*) in texts

Example: (ex:john, ex:father-of, ex:bill)

(here, **ex:** is an abbreviation for an <http://...> prefix, see below)

Can be seen as a first-order logic **fact**

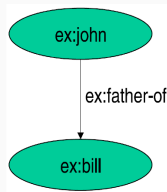
ex:father-of(ex:john, ex:bill)

RDF Graph Representation

RDF graph (set of triples) is often visualised as a labelled directed multi-graph

- nodes: subjects and objects
- edges (i.e., edges labels): predicates

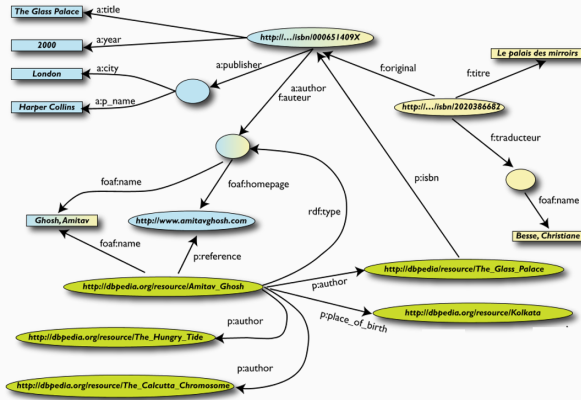
Example: (ex:john, ex:father-of, ex:bill)
can be seen as the graph on the right



Remember the mismatch:

edges can also be nodes (rare in practice)

RDF graph: More complex example



Here and usually:

- resources are **filled-in ovals**
- blank nodes are **empty ovals**
- literals are **boxes**

A resource may be an identifier of

- Web page (e.g., `<http://www.w3.org>`)
- person (e.g., `<http://www.w3.org/People/Berners-Lee/>`)
- book (e.g., `urn:isbn:4-534-34674-4`)
- anything else denoted with a URI (or IRI, a Unicode version of URI)

IRI (and URI) is an identifier and not a location on the Web

RDF allows making statements about resources:

```
(<http://www.w3.org/People/Berners-Lee/>, <http://www.w3.org/HasName>, "Tim")  
(urn:isbn:0-345-33971-1, <http://www.w3.org/Has-Author>, "John")
```


Uniform Resource Identifier (URI):

a **string of characters** used to identify a name or a resource on the Web

URI is **URL** or **URN**:

- Uniform Resource Name (URN) defines an item's identity

Example: URN `urn:isbn:4-534-34674-4` is a URI specifying the identifier system (ISBN) and the unique reference within the system

- Uniform Resource Locator (URL) provides a method for finding the resource

Example: URL `<https://www.uio.no/studier/emner/matnat/ifi/IN3020>` identifies a resource (IN3020's home page) and implies that a representation of that resource is obtainable via the HTTP link

IRI: a Unicode version of **URI**

Literal can be

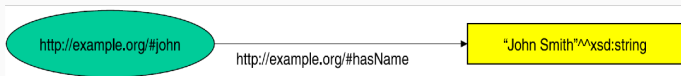
- **plain literal:**

sequence of symbols (e.g., “any text”),
optionally followed by language tag
(e.g. “Hello, how are you?”@en-GB)

- **typed literal:**

(e.g., “hello”^^xsd:string and “1”^^xsd:integer)
recommended datatypes are XML Schema datatypes
xsd:string, xsd:integer, xsd:float, xsd:boolean, etc.

Literal can be only **the object** of a triple, for example:



RDF elements: Blank nodes

Blank nodes are nodes without a URI; used for

- unnamed resources
- more complex-structured data
(RDF containers, reification, OWL anonymous classes, etc.)

Usually written with `_:` prefix

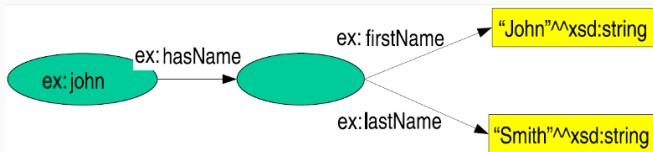
There may be different blank nodes (different from SQL NULLs)

Example:

```
(ex:john, ex:hasName, _:johnsname)
```

```
(_:johnsname, ex:firstName, "John"^^xsd:string)
```

```
(_:johnsname, ex:lastName, "Smith"^^xsd:string)
```



RDF containers:

- used to add more structure to the data
(surrogate, no semantics)
- group objects for the same subject and predicate:

Motivating examples:

“The lecture is attended by John, Mary and Chris (without specified order)”
called `bag`

“RDF-Concepts are edited by Graham and Jeremy (in that order)”
called `seq`

*“The source code for the application may be found at at least one of
ftp1.example.org, ftp2.example.org, ftp3.example.org”*
called `alt`

Three types of containers:

- `rdf:Bag`: unordered set of items
- `rdf:Seq`: ordered set of items
- `rdf:Alt`: set of alternatives

Represented by a blank node `_:cont` connected to `rdf:XXX`
by a triple `<_:cont, rdf:type, rdf:XXX>`

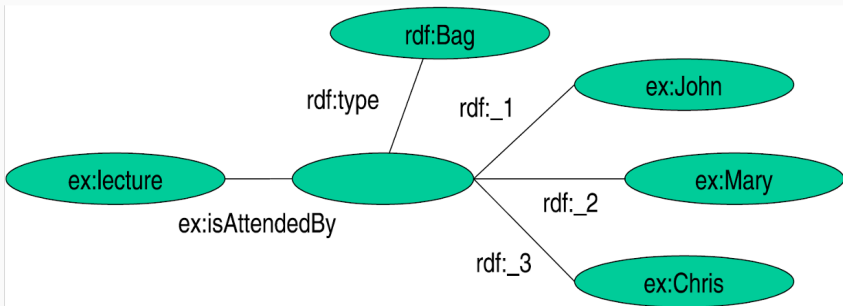
Items in the container are linked to `_:cont` by `rdf:_1`, `rdf:_2`, ...

Limitations:

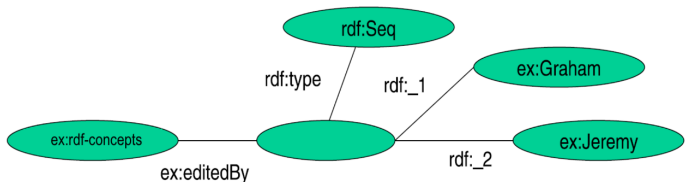
- semantics of the container is **up to the application**
- what about closed sets (e.g., how do we know whether Graham and Jeremy are the **only editors** of RDF-Concepts)?
Can be represented by **RDF collections** (see below)

RDF Bag container: Example

*“The lecture is attended by John, Mary and Chris
(without specified order)”*

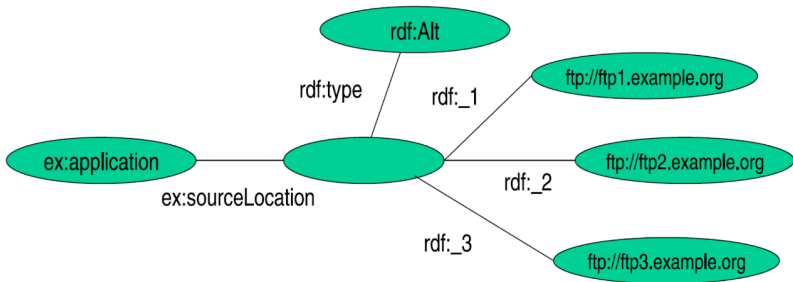


“RDF-Concepts are edited by Graham and Jeremy (in that order)”



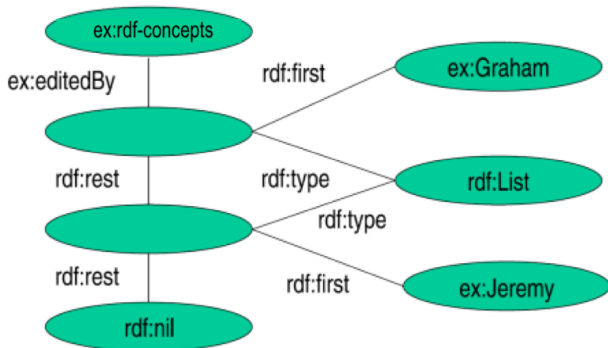
RDF Alt container: Example

“The source code for the application may be found at at least one of ftp1.example.org, ftp2.example.org, ftp3.example.org”



RDF collections: Example

*“RDF-Concepts are edited by Graham and Jeremy
(in that order and nobody else)”*



Reification: Idea

Reification: statements about statements (also more structure)

Example: *“John’s name is John Smith”* normally represented by
(`ex:john`, `ex:hasName`, *“John Smith”*)

Example: *“Mary claims that John’s name is John Smith”*

We need to relate a **URI to the triple**, which is **not possible** naturally

Reify (i.e., transform) (`ex:john`, `ex:hasName`, *“John Smith”*) to

```
(_:myStatement, rdf:type, rdf:Statement)
```

```
(_:myStatement, rdf:subject, ex:john)
```

```
(_:myStatement, rdf:predicate, ex:hasName)
```

```
(_:myStatement, rdf:object, “John Smith”)
```

and add a triple

```
(ex:mary, ex:claims, _:myStatement)
```

Reification:

has no **formal semantics**—

that is, the original and the reified forms

are not formally equivalent

(but this can be detected by an application)

RDF uses only **binary properties**

(e.g, *“John’s name is John Smith”*)

Reification can be also used for **higher-arity properties**

(e.g, *“John takes IN3020 in 2023”*)

RDF vocabulary:

resources (IRIs) in RDF dedicated for some purpose
we have seen some (`rdf:type`, `rdf:Seq`, etc.)

In reality, RDF vocabulary is defined in the namespace:

`http://www.w3.org/1999/02/22-rdf-syntax-ns#`

Grouped into several sets:

- **Classes** (`rdf:Property`, `rdf:Statement`, `rdf:XMLLiteral`, `rdf:Seq`, `rdf:Bag`, `rdf:Alt`, `rdf>List`)
- **Properties** (`rdf:type`, `rdf:subject`, `rdf:predicate`, `rdf:object`, `rdf:first`, `rdf:rest`, `rdf:_n`, `rdf:value`)
- **Other** (`rdf:nil`)

RDF Vocabulary:

- formally, there is **nothing special** about these IRI's
- may have desired assumed meaning,
which can be used in **applications**

For example, `rdf:type` is used for **typing**:

(`A`, `rdf:type`, `B`) means "*A belongs to class B*"

`rdf:Seq`, `rdf:Bag`, `rdf:Alt`, `rdf>List` mean as explained **above**

However, every RDF graph **implicitly contains** several triples:

- (`P`, `rdf:type`, `rdf:Property`) for each *P* in the **property position**
- (`rdf:type`, `rdf:type`, `rdf:Property`)
- ...

RDF has several **file formats** (i.e., syntaxes understood by systems)

- Turtle
- XML
- N-Triples (similar to **Turtle**)
- JSON-LD
- (my (s, p, o) format above does not count,
it is only for humans)

Next we briefly look at **some** of them

Simple Turtle is most **direct** syntax:

- full triples (blanks between elements, dot-separated)
- full IRIs, **enclosed** in < and >
- readable, but **wordy**

Example:

```
<http://example.org/#john> <http://.../vcard-rdf/3.0#FN> "John Smith" .

<http://example.org/#john> <http://.../vcard-rdf/3.0#N> :_X1 .
_:X1 <http://.../vcard-rdf/3.0#Given> "John" .
_:X1 <http://.../vcard-rdf/3.0#Family> "Smith" .

<http://example.org/#john> <http://example.org/#hasAge> "32" .

<http://example.org/#john> <http://example.org/#marriedTo> <http://example.org/#mary>

<http://example.org/#mary> <http://.../vcard-rdf/3.0#FN> "Mary Smith" .

<http://example.org/#mary> <http://.../vcard-rdf/3.0#N> :_X2 .
_:X2 <http://.../vcard-rdf/3.0#Given> "Mary" .
_:X2 <http://.../vcard-rdf/3.0#Family> "Smith" .

<http://example.org/#mary> <http://example.org/#hasAge> "29" .
```

Turtle also allows for some shortcuts, such as

- @base for the base IRI: may be omitted in the graph after
- @prefix introduces an abbreviation for a prefix
- ; means that in the following triple the subject is omitted and taken from the previous triple
- [] introduces an unlabelled blank node

Example:

```
@base <http://example.org/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

<#john>
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  <#hasAge> 32 ;
  <#marriedTo> <#mary> .
<#mary>
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  <#hasAge> 29 .
```


XML syntax is more wordy

Example (only):

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:ex="http://example.org">
  <rdf:Description rdf:about=ex:john>
    <foaf:name>John Smith</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about=ex:marry>
    <foaf:name>Marry Smith</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

(Note: this is a different graph)

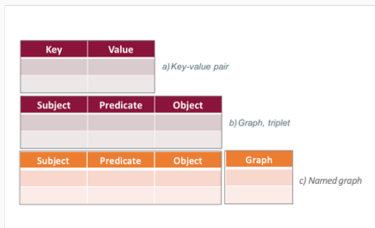
See <https://www.w3.org/TR/rdf-syntax-grammar/> for further details

RDF Named graphs

Sometimes we need to work with **several** RDF graphs
(in one RDF store)

To access them (e.g., in SPARQL), we can give them **names**
a **name** is also a **IRI**

Conceptually, we can see a triple in a named graph
as a **quadruple** of IRIs:



(We **omit** the details)

2.2. Implicit knowledge in RDF: RDFS and OWL

RDF Vocabulary is just a convention for usage of some URIs
(no special meaning)

RDFS and OWL are technologies to encode **implicit knowledge**:

- some URIs (vocabulary) have special meaning
- explicit triples with these URIs in an RDF graph imply other implicit triples (i.e., knowledge) according to specific rules

Example: graph with

`(John, rdf:type, #Student), (#Student, rdfs:subClassOf, #Person)`

implicitly contains (under RDFS entailment) `(John, rdf:type, #Person)`
according to the RDFS rule

$$\frac{(A, \text{rdf:type}, \text{Class1}) \quad (\text{Class1}, \text{rdfs:subClassOf}, \text{Class2})}{(A, \text{rdf:type}, \text{Class2})}$$

RDFS (RDF schema):

- W3C standard (see <https://www.w3.org/TR/rdf-schema/>)
- Systems working with RDF (e.g., SPARQL) can be customised to work under **RDFS entailment regime** (i.e., to take implicit triples into account)
- IN4060 – Semantic Technologies for more details

RDFS vocabulary:

RDFS Classes

- `rdfs:Resource`
- `rdfs:Class`
- `rdfs:Literal`
- `rdfs:Datatype`
- `rdfs:Container`
- `rdfs:ContainerMembershipProperty`

RDFS Properties

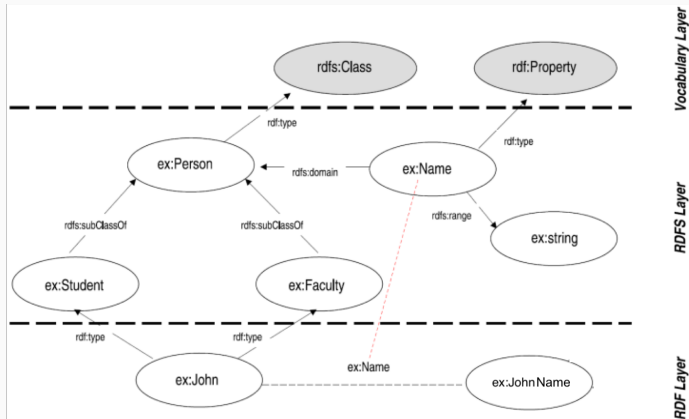
- `rdfs:domain`
- `rdfs:range`
- `rdfs:subPropertyOf`
- `rdfs:subClassOf`
- `rdfs:member`
- `rdfs:seeAlso`
- `rdfs:isDefinedBy`
- `rdfs:comment`
- `rdfs:label`

RDF vocabulary is also used in **RDFS rules**

(but have special meaning here, contrary to RDF)

RDFS entailment: Example

RDFS rules over graph



entail, for example,

`(ex:John, rdfs:type, ex:Person)`

`(ex:JohnName, rdfs:type, ex:String)`

RDFS entailment is rather simple

OWL (Web Ontology Language) goes **further** in such knowledge:

- allows for **complex** entailment rules (in fact, rule patterns) based on **Description Logics** (DLs, several versions)
- for **example** OWL implies the rule (in DL syntax)
 $\text{Student} \sqcap \text{HasIFIoffice} \sqsubseteq \text{IFIStudent}$, which means
'Every student who has an IFI office is an IFI student'
- uses **OWL vocabulary** (with `owl:` prefix)
that allows to encode such entailments
- **entailment** (called **DL reasoning**) are may be **complex**
- W3C standard (<https://www.w3.org/TR/owl2-overview/>)
- several systems: **HermiT**, **ELK**, **KAON**, etc.
- **IN4060** – **Semantic Technologies** for more details

2.3. Query language for RDF: SPARQL

Relational databases:

a **well-established** technology to store and query data
has well-known **advantages** and **disadvantages**
over other approaches to manage data

Can we **translate** (ideally, automatically) **RDF graphs** to **relations**
and use **SQL** for querying the data?

Yes, but there are some **disadvantages**

which we will see on an **example RDF graph**:

```
(person12345, rdf:type, uni:lecturer)
```

```
(person12345, uni:name, "Joe Doe")
```

```
(uni:lecturer, uni:title, "University Professor")
```

Storing RDF data in relational form: Normalised approach

Possible approach: normalise data into relation(s)

In our example,

```
(person12345, rdf:type, uni:lecturer)
```

```
(person12345, uni:name, "Joe Doe")
```

```
(uni:lecturer, uni:title, "University Professor")
```

translates to

Lecturer		
id	name	title
person12345	Joe Doe	University Professor

Advantages:

Querying is standard; for example, *'Find names of all lecturers'* can be written in SQL as `SELECT name FROM Lecturer`

Drawbacks:

Each new type of content requires a new table

Not scalable, not dynamic, not based on the RDF principles (triples)

Storing RDF data in relational form: Less normalised

Another possible approach:

relations for triples with IDs, Resources, Literals, etc.

In our **example**, the triples translate to

Statement				Resources		Literals	
Subject	Predicate	ObjectURI	ObjectLiteral	Id	URI	Id	Value
101	102	103	null	101	21345	201	Joe Doe
101	104	null	201	102	rdf:type	202	University Professor
101	105	null	202	103	uni:lecturer
...	104

Advantages:

flexible in adding new statements **dynamically** without changing the schema

Drawbacks:

using SQL for querying may be very inefficient due to a lot of joins
for **example**, *'Find names of all lecturers'* requires 5 joins

SPARQL:

- **native** query language for RDF data
- uses **SQL-like** syntax for query structure
- uses **Turtle-like** syntax for triples (possibly with variables)
- avoids the **drawbacks** of possible translations to **SQL**
- (**in one sentence**:
Looks **like SQL** over one **Triple** table,
but with **several differences** in details)

Example:

```
SELECT ?name
WHERE { ?s <http://example.org/uni/name> ?name .
       ?s rdf:type <http://example.org/uni/lecturer> }
```

In SPARQL, '?' is a **variable** prefix

SPARQL query elements

Basic SPARQL SELECT query structure:

(PREFIX ...) *

- prefix mechanism for abbreviating URIs, as in Turtle

SELECT [DISTINCT | REDUCED] ...

- identifies the variables to be returned in the query answer

[FROM [NAMED] ...]

- the graph to be queried (if not default)

WHERE ...

- query pattern to match (may be complex)

[ORDER BY ...] [LIMIT ...] [OFFSET ...]

- number and order of answers, as in SQL

Besides SELECT, there are ASK, CONSTRUCT, etc.,
which we will look at the end of the lecture

More about some SPARQL Keywords

PREFIX:

- specifies a namespace as in **Turtle**
- may be none, one, or many
- **Example:** PREFIX uni: <http://example.org/uni/>

SELECT DISTINCT and SELECT REDUCED:

- normally, SPARQL gives a bag of mappings
(i.e., variable assignments, see below)
- **DISTINCT** does duplicate elimination, as in SQL
- **REDUCED** is flexible: leave the system to decide eliminate or not

ORDER BY, LIMIT, OFFSET:

- order the answers and specifies the fraction, as in SQL
- may be several order comparators, possibly with **ASC**, **DESC** modifiers
- **Example:**

```
SELECT ?name, ?age WHERE ...  
ORDER BY ?name, DESC(?age) LIMIT 5 OFFSET 10
```

SPARQL Example: Subject and object retrieval

Query *'Return the full names of all people in the graph'*:

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?fullName
WHERE { ?x vCard:FN ?fullName }
```

over **RDF** graph

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

gives

```
?fullName
```

```
=====
```

```
"John Smith"
```

```
"Mary Smith"
```

Essentially, **selection** and **projection**

SPARQL Example: Property retrieval

Query *'Return the relations between John and Mary':*

```
PREFIX ex: <http://example.org/#>
```

```
SELECT ?p
```

```
WHERE { ex:john ?p ex:mary }
```

over **RDF** graph

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

gives

?p

=====

<http://example.org/#marriedTo>

Essentially, **selection** and **projection**

SPARQL Example: Several-triple patterns

Query *'Return the spouse of a person by the name of John Smith':*

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
PREFIX ex: <http://example.org/#>
```

```
SELECT ?y
```

```
WHERE { ?x vCard:FN "John Smith" . ?x ex:marriedTo ?y }
```

over **RDF graph**

gives

?y

=====

<http://example.org/#mary>

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

Essentially, **selection**, **projection**, and **join**

SPARQL Example: Several answer variables, blank nodes

Query *'Return the names with their first names'*:

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE { ?x vCard:N ?name . ?name vCard:Given ?firstName }
```

over **RDF graph**

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

gives

```
?name ?firstName
```

```
=====
```

```
_:a      "John"
```

```
_:b      "Mary"
```

Blank nodes (in RDF graphs) are treated as **usual** URIs

SPARQL Example: Filter

Query 'Return all people over 30':

```
PREFIX ex: <http://example.org/#>
```

```
SELECT ?x
```

```
WHERE { ?x hasAge ?age . FILTER(?age > 30) }
```

over **RDF graph**

gives

?x

=====

<http://example.org/#john>

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

FILTER is a part of **WHERE** clause, essentially **complex selection**

SPARQL Example: Optional matching

Query *'Return all people and their spouses, if we know about any'*:

```
PREFIX ex: <http://example.org/#>
```

```
SELECT ?person, ?spouse
```

```
WHERE { ?person ex:hasAge ?age .
```

```
        OPTIONAL { ?person ex:marriedTo ?spouse } }
```

over **RDF** graph

gives

```
?person ?spouse
```

```
=====
```

```
<.../#mary>
```

```
<.../#john> <.../#mary>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

OPTIONAL sub-clauses may be complex

(e.g., several triples, nested optional with **bottom-up** evaluation)

SPARQL WHERE queries: Summary

Syntax of SPARQL WHERE clause:

- triple pattern as base case (a triple possibly with variables)
- . as 'join', similar AND in SQL
- more complex conditions via FILTER
- optional matching via OPTIONAL (no direct SQL analog)
- bag union via UNION (does not require same sets of variables, cf. SQL)
- property paths (reachability in a graph, difficult to express in SQL)
- aggregates, subqueries, etc.

Semantics SPARQL WHERE queries:

- RDF graph as input, a multiset (bag) of mappings as output
- a mapping is an assignment of RDF terms to variables
(may have different variables in mapping domains in the same multiset)
- query evaluation may be formally defined using SPARQL algebra
(manipulates bags of mappings,
evaluated bottom-up: from simple sub-queries to more complex)

SPARQL ASK Example: Testing if a solution exists

Query *'Are there any married persons'*:

```
PREFIX ex: <http://example.org/#>
```

```
ASK { ?person ex:marriedTo ?spouse }
```

over **RDF** graph

gives

=====

yes

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

ASK ... is essentially SELECT (no variables) WHERE ...

SPARQL CONSTRUCT Example: Building RDF graphs

Query *'Rewrite the naming triples from vCard to foaf'*:

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
CONSTRUCT { ?x foaf:name ?name }
```

```
WHERE { ?x vCard:FN ?name }
```

over RDF graph

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

gives RDF graph

```
ex.john foaf:name "John Smith".
```

```
ex.marry foaf:name "Marry Smith"
```

CONSTRUCT creates a new graph

There are several **triplestores**,
most of which include **SPARQL engines**:

- AllegroGraph
- OpenLink Virtuoso
- GraphDB
- Apache Jena
- Oracle Spatial and Graph DB
- RDFox
- etc.

See details related to scalability of triplestores at
<https://www.w3.org/wiki/LargeTripleStores>

3. Property path graphs (Neo4j, Cypher)

3.1. Neo4j (and GQL) data model

(Labelled) graph databases

Labelled graphs:

natural to describe data and traverse through it

- each **node** has an **inner structure** describing its properties
- the **edges** indicate **relationships** between the nodes
- the **edges** can carry **information** in the same way as the nodes

Data in the labelled-graph form: can be **schema-free**

- new nodes and edges (with new inner structures) can be introduced **dynamically**
- existing nodes and edges can be **expanded** with new properties

Search: Specify how the graph should be **navigated**

- the graph is **traversed** via pointers to neighbouring nodes
(the traverse requires **no** indexes and no join operations)

Neo4j (<https://neo4j.com>):

- a **native** graph database
- 'whiteboard friendly'
- **schemaless** – no need to define any structure in advance
- **query** language: **Cypher**
 - **declarative**, pattern-based
 - **transaction** support
- **scalability** (support for clusters)
- **examples** of use: **eBay**, **HP**
- **open-source** under **GPL**

Easiest to get started is via the Neo4j Sandbox:

<https://neo4j.com/sandbox>

Alternatively, download and install locally the Neo4j desktop:

<https://neo4j.com/download/neo4j-desktop/>

Neo4j data model: Property Graphs

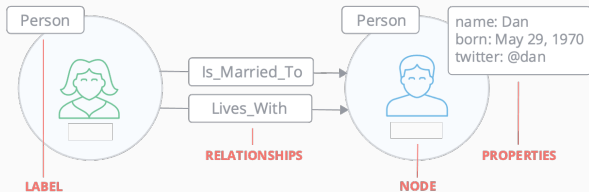
Node: can have **any** number (none, one or several) **labels**

Relationships:

- **directed** edges between two nodes
- two nodes can have **several** relationships between
- each relationship has **exactly one type**

Properties:

- **key-value pairs**
- can be attached to **both** nodes and relationships (any number)
- **key** is a character string
- **value** is from a **base datatype** (e.g., **int**, **char**) or an **array** over a base datatype (e.g., **int[]**, **char[]**)



3.2 Cypher query language

Cypher Graph patterns: Intro

The strength of a **property graph** lies in its ability to encode patterns of connected nodes and relationships


Cypher is strongly based on **patterns** (cf. SPARQL patterns)

- patterns are used to match desired **graph structures**
- **simple pattern** connects two nodes by a single relationship
example: Person LIVES_IN City
- **complex patterns** can express arbitrarily complex concepts (i.e., use many nodes and relationships)
example: Person LIVES_IN City IS_PART_OF Country

Cypher represents graph-related patterns using clauses and keywords for **example**, MATCH, WHERE and DELETE

All the **details** at <https://neo4j.com/docs/cypher-manual>

Alternatively, you can start with the infamous Cypher Refcard


Neo4j Cypher Refcard 4.4

Legend

- Read
- Write
- General
- Functions
- Schema
- Performance
- Multidatabase
- Security

Syntax

Read query structure

```
[LIMIT]
[OPTIONAL MATCH WHERE]
[MATCH [ORDER BY]] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

MATCH [Ⓒ]

```
MATCH (n:Person)-[:KNOWS]->(n:Person)
WHERE n.name = 'Alice'
Node patterns can contain labels and properties.
MATCH (x)->(n)
Any pattern can be used in MATCH.
MATCH (n {name: 'Alice'})->(n)
Patterns with node properties.
```

RETURN [Ⓒ]

```
RETURN *
Return the value of all variables.
RETURN n AS columnName
Use alias for result column name.
RETURN DISTINCT n
Return unique rows.
ORDER BY n.property
Sort the result.
ORDER BY n.property DESC
Sort the result in descending order.
SKIP $skipNumber
Skip a number of results.
LIMIT $limitNumber
Limit the number of results.
SKIP $skipNumber LIMIT $limitNumber
Skip results at the top and limit the number of results.
RETURN count(*)
The number of matching rows. See Aggregating Functions for more.
```

WITH [Ⓒ]

```
MATCH (user)-[:FRIENDS]-(:friend)
WHERE user.name = $name
WITH user, count(:friend) AS friends
WHERE friends > 10
RETURN user
The WITH SYNTAX is similar to RETURN. It separates query parts explicitly, allowing you to declare which variables to carry over to the next part.
MATCH (user)-[:FRIENDS]-(:friend)
```

Operators [Ⓒ]

General	DISTINCT, ..[]
Mathematical	+, -, *, /, %, ^
Comparison	<, <=, >, >=, =, IS NULL, IS NOT NULL
Boolean	AND, OR, XOR, NOT
String	+
List	+, IN, [x], [x .. y]
Regular Expression	~
String matching	STARTS WITH, ENDS WITH, CONTAINS

null [Ⓒ]

- null is used to represent missing/undefined values.
- null is not equal to null. Not knowing two values does not imply that they are the same value. So the expression null = null yields null and not true. To check if an expression is null, use IS NULL.
- Arithmetic expressions, comparisons and function calls (except coalesce) will return null if any argument is null.
- An attempt to access a missing element in a list or a pattern that doesn't exist yields null.
- In OPTIONAL MATCH clauses, nulls will be used for missing parts of the pattern.

Patterns [Ⓒ]

```
(n:Person)
Node with Person label.
```

USE [Ⓒ]

```
USE myDatabase
Select myDatabase to execute query, or query part, against.
USE next;
MATCH (n:Person)-[:KNOWS]->(n:Person)
WHERE n.name = 'Alice'
MATCH query executed against next; database.
```

SHOW FUNCTIONS and PROCEDURES [Ⓒ]

```
SHOW FUNCTIONS
Listing all available functions.
SHOW PROCEDURES EXECUTABLE YIELD name
List all procedures that can be executed by the current user and return only the name of the procedures.
```

SHOW and TERMINATE TRANSACTIONS [Ⓒ]

```
SHOW TRANSACTIONS
Listing all available transactions.
TERMINATE TRANSACTIONS 'neo4j-transaction-42'
Terminate the transaction with ID neo4j-transaction-42.
```

Labels

```
CREATE (n:Person {name: 'Solve'})
Create a node with label and property.
MATCH (n:Person {name: 'Solve'})
Matches or creates unique node(s) with the label and property.
SET n:Developer:Employee
Add label(s) to a node.
.....
```


Node syntax

Nodes are represented in Cypher using a pair of parentheses:

- `()` an **anonymous, uncharacterised** node
- `(varName)` a node labelled by a **variable** (restricted to a **single** statement)
- `(:label)` restrict the matches of the node by a **specific label**
- `{key1:value1, ... , keyn:valuen}` restrict the matches of the node by a **specific properties**

Examples:

```
()  
(matrix)  
(:Movie)  
(matrix:Movie)  
(matrix:Movie {title: "The Matrix"})  
(matrix:Movie {title: "The Matrix", released: 1997})
```

Relationship syntax

Relationships are represented with dashes:

- `-- relationships` (direction and types are unrestricted)
- `<--`, `-->` `directed relationships`
- `[...]` to specify `details`
(`variables`, `relationship types`, `properties`)

Examples:

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Patterns are expressed by combining nodes and relationships
node relationship node

Examples:

```
(keanu:Person:Actor {name: "Keanu Reeves"} )  
-[role:ACTED_IN {roles: ["Neo"]} ]->  
(matrix:Movie {title: "The Matrix"} )
```

Cypher statements typically have multiple clauses, each clause performs a specific task, such as

- create and match patterns in the graph
- filter, project, sort, or paginate results
- compose partial statements

Creating data

The simplest clause is `CREATE`

```
CREATE (:Movie { title:"The Matrix",released:1997 })
```

```
+-----+  
| No data returned. |  
+-----+
```

```
Nodes created: 1  
Properties set: 2  
Labels added: 1
```

To return the created (or retrieved) data the `RETURN` clause is used
(refers to the variable assigned to the pattern elements)

```
CREATE (p:Person { name:"Keanu Reeves", born:1964 })  
RETURN p
```

```
+-----+  
| p |  
+-----+  
| Node[1]{name:"Keanu Reeves",born:1964} |  
+-----+
```

```
1 row  
Nodes created: 1  
Properties set: 2  
Labels added: 1
```

We can **create** more complex structures

```
CREATE (a:Person { name:"Tom Hanks",  
  born:1956 })-[r:ACTED_IN { roles: ["Forrest"]}]->(m:Movie { title:"Forrest Gump",released:1994 })  
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[ :DIRECTED]->(m)  
RETURN a,d,r,m
```

Usually, we want to **connect** new data to **existing** structures

Requires us to **find** existing patterns in graph data

Matching patterns

Matching patterns is done using the **MATCH** statement, by passing the patterns describing what to look for

MATCH statement searches for the specified patterns and return one row per successful pattern match

```
MATCH (p:Person { name:"Tom Hanks" })-[r:ACTED_IN]->(m:Movie)
RETURN m.title, r.roles
```

m.title	r.roles
"Forrest Gump"	["Forrest"]

1 row

It is possible to **attach** structures to the graph by combining **MATCH** and **CREATE**

```
MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]>(m)
RETURN p,r,m
```

Completing patterns

`MERGE` checks for the **existence** of data first before creating it
defines a pattern to be found or created
(can provide **additional** properties to set `ON CREATE`)

```
MERGE (m:Movie { title:"Cloud Atlas" })  
ON CREATE SET m.released = 2012  
RETURN m
```

`MERGE` can also **assert** that a relationship is only created **once**

```
MATCH (m:Movie { title:"Cloud Atlas" })  
MATCH (p:Person { name:"Tom Hanks" })  
MERGE (p)-[r:ACTED_IN]->(m)  
ON CREATE SET r.roles = ['Zachry']  
RETURN p,r,m
```


Filtering results

Filtering conditions are expressed in a `WHERE` clause

(here, `=~` is regular expression matching operator)

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000 OR "Neo" IN r.roles
RETURN p,r,m
```

Allows to use **Boolean expressions** with `AND`, `OR`, `XOR` and `NOT`

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

Returning results

The `RETURN` clause can return nodes, relations, and **expressions**

Examples of simple **expressions**:

- **values** of keys from key-value pairs: **numbers**, **arrays**, etc.
- **functions** over other expressions:
 `a + b`, `length(array)`, `toInteger("12")`, etc.

Can use **expression AS alias** to improve **readability**

`DISTINCT` indicate unique results in `RETURN`

```
MATCH (p:Person)
RETURN p, p.name AS name, toUpper(p.name), coalesce(p.nickname,"n/a") AS nickname, { name: p.name,
  label:head(labels(p))} AS person
```

Aggregating information, ordering and pagination

Aggregation over final results can be used in the `RETURN` clause
common aggregation functions are supported
(e.g., `count`, `sum`, `avg`, `min`, and `max`)

```
MATCH (:Person)  
RETURN count(*) AS people
```

```
+-----+  
| people |  
+-----+  
| 3      |  
+-----+  
1 row
```

Ordering by `ORDER BY` expression
(optionally with `ASC/DESC` modifiers)

Pagination by `SKIP` offset `LIMIT` count

Other features: `UNION`, `WITH`, etc.

Graph tasks (reminder)

Used to compute **metrics** for **graphs**, **nodes**, or **relationships**

Provide **insights** on

- **nodes** and **pairs** of nodes (shortest path, etc.)
- relevant **entities** in the graph (centralities, ranking),
- inherent **structures** (community-detection, graph-partitioning, clustering)

Many of these tasks have high (worst-case) **complexity**

- **iterative approaches** frequently traverse the graph for the computation using random walks, breadth-first or depth-first searches, or pattern matching
- **optimised algorithms** utilise certain structures of the graph, recall already explored parts, and parallelise operations

Neo4j has highly efficient built-in **library** for many graph tasks
(**distinctive feature** in comparison to other approaches)

Path finding:

find the shortest path or evaluate the availability and quality of routes (minimum-weight spanning tree, random walk, all-pairs and single-source shortest path via A* and Yen's K-shortest paths)

Centralities:

determine the importance of distinct nodes in a network (PageRank, Betweenness Centrality, Closeness Centrality)

Community detection:

evaluate how a group is clustered or partitioned, as well as its tendency to strengthen or break apart (Louvain, label propagation, connected components, triangle count & clustering coefficient)

Example: Shortest Path

Calculates the shortest (weighted) path between a pair of nodes
(Dijkstra's algorithm is the most well-known approach)

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath.stream(start, end, 'cost')
YIELD nodeId, cost
MATCH (other:Loc) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```

name	cost
"A"	0.0
"B"	50.0
"D"	90.0
"E"	120.0
"F"	160.0

Often used to finding **directions** between physical locations
(e.g., [Google maps](#))

Can also be used in **social networks**
to find the **degrees of separation** between people

Removing and modifying data

DELETE: **deletes** nodes, relationships or paths

DETACH DELETE: **deletes** a node and all relationships to or from it

```
MATCH (n:Person { name: 'UNKNOWN' })
DELETE n
```

```
MATCH (n { name: 'Andy' })
DETACH DELETE n
```

```
MATCH (n { name: 'Andy' })-[r:KNOWS]->()
DELETE r
```

REMOVE: **removes** key-value pairs from nodes and relationships, and removes labels from nodes

```
MATCH (a { name: 'Andy' })
REMOVE a.age
RETURN a.name, a.age
```

```
MATCH (n { name: 'Peter' })
REMOVE n:German
RETURN n.name, labels(n)
```

SET: **updates** labels and values of keys

```
MATCH (n { name: 'Andy' })
SET n.surname = 'Taylor'
RETURN n.name, n.surname
```

```
MATCH (n { name: 'George' })
SET n:Swedish:Bossman
RETURN n.name, labels(n) AS labels
```

Graph vs. relational models

Graph model vs. Relational model:

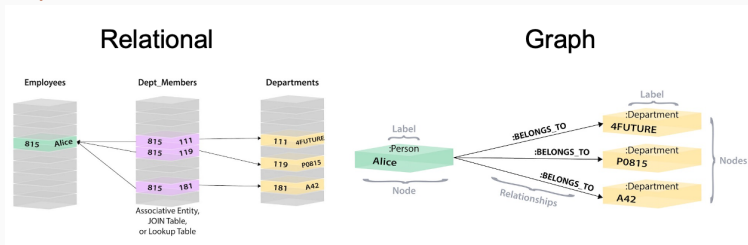
- traversing a graph is much **cheaper** than joins;
 uses direct **pointers** to neighbouring nodes
- workload is **shifted** from query execution
 to data insertion and maintenance
- '**dynamic**' **schema** make it simpler to use for not-experts

Signs of managing **highly-connected** data with a **relational database**

- large number of **JOINS**
- numerous **self-JOINS** (or **recursive JOINS**)
- frequent **schema changes**
- **slow-running** queries (despite extensive tuning)
- **pre-computing** the results

From Relational to Graph DBs

Example:



SQL

```
SELECT firstName, lastName FROM Person  
LEFT JOIN Dept_Member  
ON Person.personId = Dept_Member.personId  
LEFT JOIN Department  
ON Department.deptId = Dept_Member.deptId  
WHERE Department.deptName = "IT Department"
```

Cypher

```
MATCH (p:Person)-[:WORKS_FOR]->(d:Department)  
WHERE d.name = "IT Department"  
RETURN p.firstName,p.lastName
```

Comparison:

<https://neo4j.com/developer/graph-db-vs-rdbms/>

Tips for translation:

<https://neo4j.com/developer/relational-to-graph-modeling/>

Multi-model DB is

a database that consists of **different** data storage mechanisms (e.g., **relational**, **document**, **key/value**, **graph**):

- all in one **database engine**
- unifying **query language** and **API**
- cover all supported **data models** and
even allow for **mixing them** in a **single query**

Multi-model databases: Examples

- ArangoDB: document (JSON), graph, key-value
- Cosmos DB: document, table, key-value, JSON, SQL
- CouchBase: relational (SQL), document
- CrateDB: relational (SQL), document (Lucene)
- MarkLogic: document (XML and JSON),
graph (RDF with OWL/RDFS), text, geospatial,
relational (SQL)
- OrientDB: document (JSON), graph, key-value, text,
geospatial, binary, reactive, relational (SQL)
- Datastax: key-value, tabular, graph
- etc.

What have we learned?

Graph databases, including RDF/SPARQL and Neo4j/Cypher

IN4020: Database Systems (2024)

Lecture 29: Database Research Trends

Leif Harald Karlsen
leifhka@ifi.uio.no

University of Oslo

6 May 2024

Lecture overview

Look at:

- ◆ trends in database research
- ◆ related courses at IFI
- ◆ related research initiatives at UiO

The Beckman report (2016)

- ◆ Available online¹
- ◆ Report from meeting(s) of top database researchers
- ◆ Meets every 5 years
- ◆ Identified big data as the defining challenge
- ◆ Identified the following trends:
 - ◆ "developing scalable data infrastructures"
 - ◆ "coping with increased diversity in both data and data management"
 - ◆ "addressing the end-to-end data-to-knowledge pipeline"
 - ◆ "responding to the adoption of cloud-based computing"
 - ◆ "accommodating the many and changing roles of individuals in the data life cycle"

¹<https://cacm.acm.org/research/the-beckman-report-on-database-research/>

Main challenge: Big data

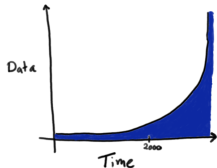
- ◆ Main characteristics: Volume, Velocity and Variety
- ◆ Need to adapt current solutions
- ◆ Many systems move to cloud
- ◆ Need completely new solutions and techniques



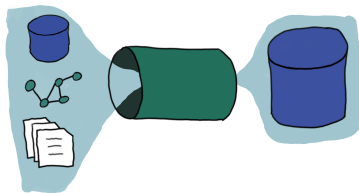
Trend: Scalable big/fast data infrastructures

Need to develop:

- ◆ more cost-aware query processing and optimization (multiple cores, large clusters)
- ◆ more specialized hardware (database machine)
- ◆ more cost-efficient storage (new types of memory)
- ◆ high-speed data-stream processing
- ◆ late-bound schemas (for un-/rarely-used data)
- ◆ new models for consistency and concurrency (NoSQL)
- ◆ new types of benchmarks (cost of ownership)



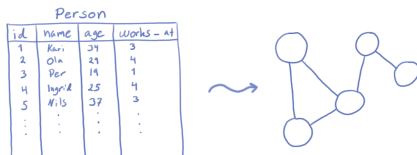
Trend: Coping with diversity in data management



Need to develop:

- ◆ better cross-platform integration (performance, abstraction)
- ◆ programming models suitable for operations on very large datasets (R, Python, etc.)
- ◆ better data processing workflows (lazy "Lego bricks")

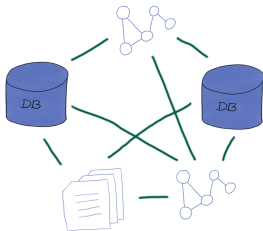
Trend: End-to-end processing of data



Need to develop:

- ◆ data-to-knowledge pipeline adapted to scale and diversity
- ◆ more diverse, (knowledge-)customizable and open source tools
- ◆ more tools for understanding data (visualization, explanation, etc.)
- ◆ knowledge bases capturing more of domain knowledge

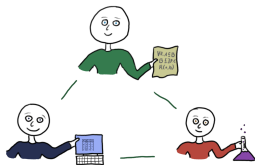
Trend: Cloud services



Need to improve:

- ◆ data elasticity (make data easier to move)
- ◆ protocols for data replication
- ◆ automatic system administration
- ◆ multitenancy (multiple users of same physical resource)
- ◆ data curation and provenance in the cloud
- ◆ support for hybrid clouds (local + cloud)

Trend: Roles of people in the data life cycle



New roles:

- ◆ Data producers: Make it easier to e.g. add metadata
- ◆ Data curators: Make it easier to crowd-source curation
- ◆ Data consumers: Make it easier to pose complex queries
- ◆ Online communities: Make it easier to share, mine, exploit data

The Seattle Report (2022)

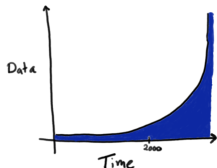
- ◆ Available online²
- ◆ Have made progress on 2016-trends, but still lots to do
- ◆ Many new developments (AI/ML, data science)
- ◆ Trends:
 - ◆ Data science as a field
 - ◆ Data governance
 - ◆ Cloud data services
 - ◆ Database engines

²<https://dl.acm.org/doi/pdf/10.1145/3524284>

Trend: Data science as a field

Need to improve:

- ◆ data-to-insight pipelines (cleaning, integration, transformation takes 80-90% of the time)
- ◆ data provenance (track data, reproduce)
- ◆ interactive data exploration at scale
- ◆ declarative programming for data
- ◆ metadata management for improved ML



Trend: Data governance

Need to improve:

- ◆ data use policy (e.g. GDPR)
- ◆ data privacy (e.g. in analysis/aggregation)
- ◆ ethical considerations (e.g. bias, discrimination)



Trend: Cloud services

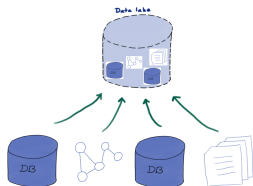
Need to improve:

- ◆ systems for serverless cloud database services (autoscaling, pay for quick allocation)
- ◆ support for disaggregation (separate storage and compute)
- ◆ support for multitenancy
- ◆ support for edge computing
- ◆ support for hybrid/multi cloud
- ◆ auto-tuning and configuration
- ◆ SaaS cloud database applications (study tradeoffs of multitenancy)

Trend: Database engines

Need to:

- ◆ support heterogeneous computing (GPU, FPGA)
- ◆ improve algorithms for distributed transactions
- ◆ make better systems for data lakes
- ◆ improve algorithms for query approximation
- ◆ improve in-database ML-inferencing
- ◆ investigate ML for auto-tuning
- ◆ improve benchmarking and reproducibility



Other relevant courses at IFI

- ◆ IN3060/4060: Semantic Web Technologies
- ◆ IN5040: Advanced Database Systems for Big Data
- ◆ IN5800: Declarative Data Engineering

Database reasearch at UiO: Examples of centers and projects

- ◆ SIRIUS: *Centre for Scalable Data Access*³ (SFI, 2015-2023)
- ◆ Respire: *Responsible Explainable Machine Learning for Sleep-related Respiratory Disorders*⁴ (2022-2027)
- ◆ Integreat: *Norwegian Centre for Knowledge-driven Machine Learning*⁵ (SFF, 2023-2033)

³<https://sirius-labs.no/>

⁴<https://www.mn.uio.no/ifi/forskning/prosjekter/respire/index.html>

⁵<https://www.integreat.no/>