

Implementation of Databases

Chapter 1: Architectures of Database Systems

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

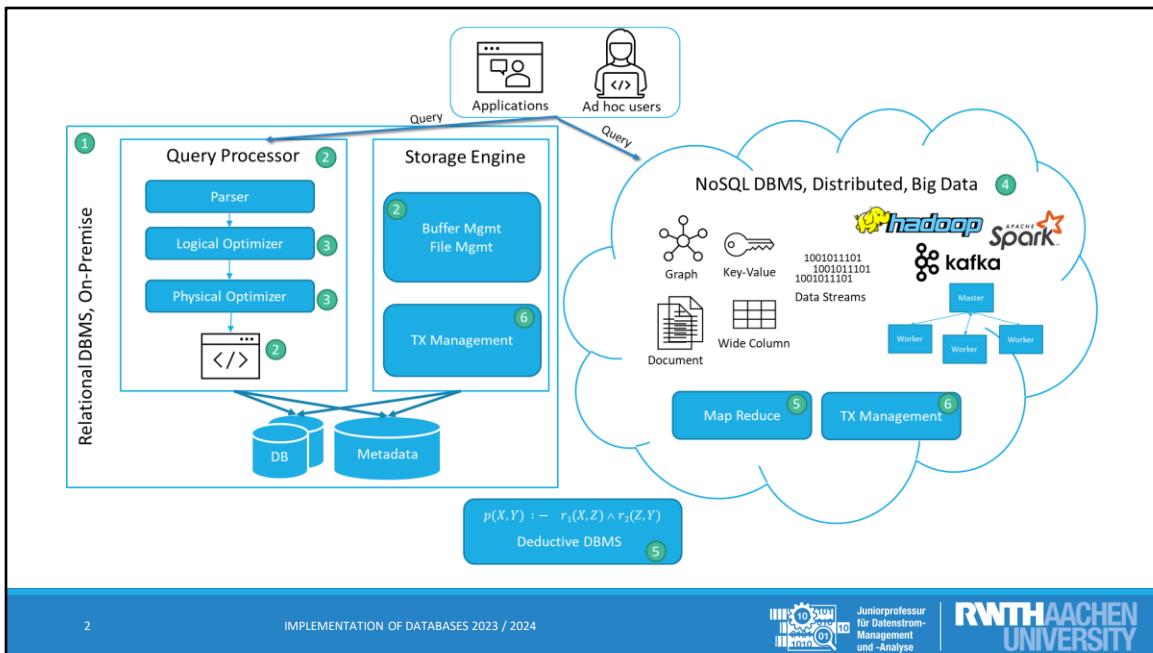
Exercises

Anastasiia Belova, M.Sc.

Soo-Yon Kim, M.Sc.



RWTHAACHEN
UNIVERSITY





1.1 Goals and Tasks of DBMS

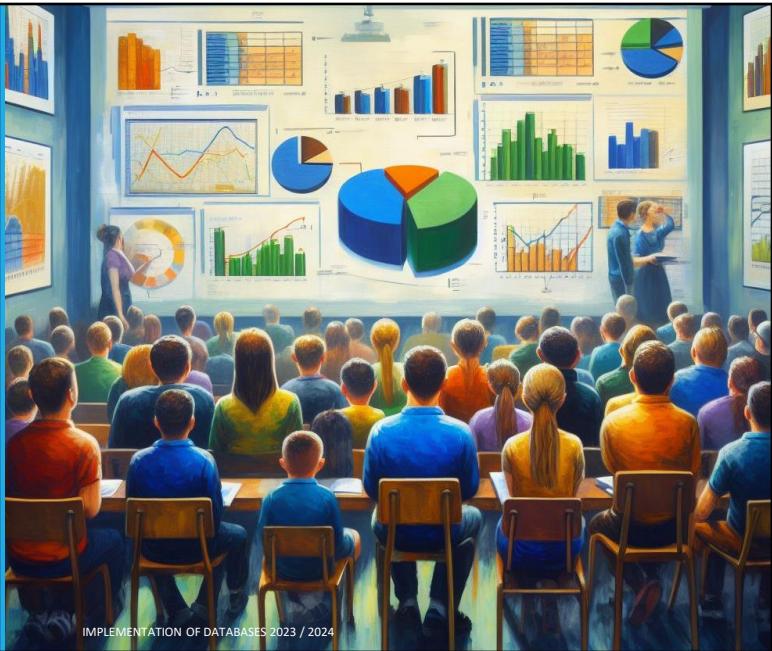
3

IMPLEMENTATION OF DATABASES 2023 / 2024

Learning Goals

At the end of this section, you will be able to

- ✓ explain, what are DBS and DBMS
- ✓ name and describe the goals and tasks of a DBMS
- ✓ discuss the relationship between software applications and DBMS



What is a Database System?



5

IMPLEMENTATION OF DATABASES 2023 / 2024



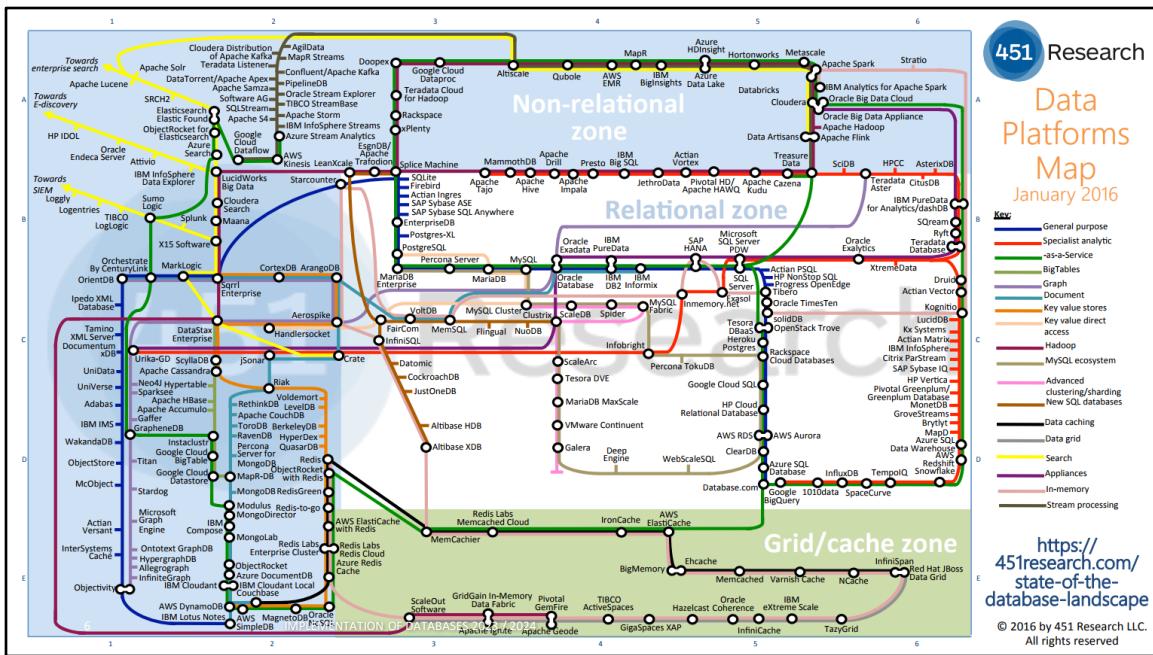
RWTHAACHEN
UNIVERSITY

Database

- Is a self-describing collection of integrated records, where a record is a representation of some physical or conceptual object.
- Represents some aspect of the real world (also universe of discourse UoD), has inherent meaning, built for a specific purpose
- A database is self-describing as it contains a description of its own structure. This description is called metadata - data about the data.
- It includes the relationships among data items, as well as including the data items themselves

DBMS

- Computerized system to create and maintain a database
- General-purpose software system to enable definition, construction, manipulation and sharing databases among users and applications
- Metadata stored in a catalog/dictionary



There are a lot of different DBMS around today. They have different characteristics, optimized for specific tasks, e.g., databases which are especially suited for time series,

graph databases suited for network analysis, object-oriented databases to store objects e.g., suited for specific data handling in engineering.

Hence, it is important to know their strengths and weaknesses and decide depending on the requirements at hand, which one to choose.

In the following we will discuss some basic principles DBMS rely on, though the implementation of the concepts may vary depending on the DBMS type.

Goals and Tasks



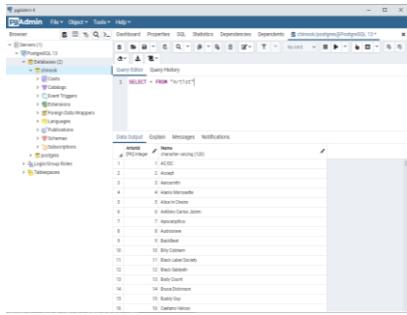
7

IMPLEMENTATION OF DATABASES 2023 / 2024



RWTH AACHEN
UNIVERSITY

Goals and Tasks - Data



Data independence

- Manage data independently of applications
- **Physical data independence:** logical schema is independent of physical structure
- **Logical data independence:** external schema (for users / applications) is independent of logical schema

Data manipulation & retrieval

- DML: Data Manipulation Language
- CRUD operations: create – **read** – update – delete

Data Independence

In very old application systems, applications and data were closely connected

- Tailored to the task and application, hard-coded
- Difficult to reuse data in different applications

Main goal of DBMS: **data independence**

- manage data independent of application (general-purpose system)
- make data accessible for different applications

(Three-level architecture on the board: External Views – Conceptual Schema – Internal Schema – Stored Databases)

- Physical DI: physical organization of database is transparent, relational schema is independent of indexes, clustering, etc.,
- Logical DI: relational views and applications are defined as derived relations on top of logical schema (the relational schema with the base relations);
logical schema might change while external schema does not need to be changed.

→ harder to achieve

Data Manipulation

What is required to achieve this?

- We need to have functions to manipulate data, for example to retrieve and select data, and to update data.
- These functions are usually provided by a **DML (data manipulation language)**
- In relational database systems, we can use SQL queries and corresponding update statements. (need CRUD)

Goals and Tasks - Consistency

Structure definition and integrity assurance

- DDL: Data Definition Language
- Data dictionary / system catalog / metadata
- Integrity conditions / assertions / constraints



Protection of databases in multi-user mode

- Transaction management
- **ACID**: Atomicity, Consistency, Isolation, Durability
- Recovery: Restart on error
- Data security and data protection



9

IMPLEMENTATION OF DATABASES 2023 / 2024



RWTH AACHEN
UNIVERSITY

Structure definition

We can define the structure of the data by a data definition language **DDL (defines metadata)**, e.g. SQL in relational systems, XML & DTD (document type definition) , is available for each data model.

It should also be possible to define integrity constraints or assertions, such as PKs and FKS.

The structure is stored in a **data dictionary or system catalogue**, which gives information about the data of the database and describes its meaning

This is **metadata**.

TX Management

- As the data can be accessed by different applications at the same time, it should also be accessible by different users at the same time.
- Therefore, a database system must have some kind of transaction management, where transactions are a set of actions initiated by a user or a system, such as reading or writing data.

- The transaction management is designed towards the ACID principle, which ensures, that the database is always in a consistent state for everyone
 - **atomicity** means: a transaction should be performed completely or not at all
→ DB is changed into a consistent new state or is the same as before
 - **consistency** means that after a transaction, the database is still in a consistent state and all integrity conditions of a DB are met by the transaction (if key constraint is violated -> TX is aborted)
 - **isolation** means that transactions running in parallel do not influence each other, they are executed if there would be not other TX, uses only persistent, consistent data
 - **durability** means that if a transaction has been executed, its changes will also survive future system crashes (also, if the change is only in the database buffer and not on disk).
- Recovery is also an important feature of a DBMS: There should also be a functionality to recover from a system crash, for example due to hardware, software or power failure
- And finally there should be a method to store data securely and protect it from unauthorized access. We will address the problem of transaction management in more detail in **chapter 6** of this course.

Goals and Tasks - Interaction

Realization of user interfaces

- Interactive end-user interface
- API: Application Programming Interface



Performance control

- Monitoring of the system load and runtime behavior
- Indexing
- Clustering / data aggregation / distribution



Realization of user interfaces

And finally, there should be some interfaces, an interactive end-user interface, and an application programming interface

Performance control

Furthermore, we should be able to control and improve the performance, for example by creating indexes and different data clusters

Sometimes one server may not be adequate to manage the amount of data or the number of requests.

DBMS and Applications



In contrast to OS,
DBMS is an
application



Databases: Consistent
non-volatile memory



Application:
Presentation of data,
data processing



Application and
DBMS often run on
different computers



Communication:
Connections between
software systems
(partially a job of OS)



ISO-OSI Reference
Model for
communication

AP	7	Application
SO	6	Presentation
	5	Session
	4	Transport
	3	Network
	2	Data Link
	1	Physical

So to conclude:

A DBMS is a general-purpose software system (itself an application) and should be independent of applications using the data it manages, while an application presents and processes the data.

Applications and DBMS run often on different computers (applications can crash the server more easily for example).

The communication between them is handled usually via standard protocols along the ISO-OSI model, partly it can be handled by the OS.

In following section, we will clarify how a system architecture can support all the mentioned tasks and how can an architecture be designed optimally for these tasks. In this context, we will get to know an architecture to consist of a set of component types and a set of interfaces between these components.

Quiz

<https://www.menti.com>
Code **3951 6372**



12





1.2 Basic Architecture of a DBMS

13

IMPLEMENTATION OF DATABASES 2023 / 2024

In the last section we have discussed the various goals and tasks of a DBMS.

In this section, we will clarify how a system architecture can support all the mentioned tasks and how can an architecture be designed optimally for these tasks.

Learning Goals

At the end of this section, you will be able to

- ✓ Explain the basic architecture of a DBMS
- ✓ Name the most important layers and interfaces of the basic and the 5-layer models and describe their tasks
- ✓ Give an example, how a query is processed throughout the layers

14



IMPLEMENTATION OF DATABASES 2023 / 2024

System Architecture Goal: Modularization



- Abstraction
- Localization
- Information hiding principle / black box
- Completeness (on an abstract level)
- Verifiability

Concepts for modularization

- Functional abstraction
- Data abstraction
- Generic modules with objects and methods

As we learned from the last section, one of most important principles for DBMS is **data independence**.

To reach this goal, the DBMS architecture needs to isolate DBMS and applications as much as possible by encapsulating components as far as possible.

One of the most important principles of DBMS design is modularization, i.e., that the different components of the system

fulfil a defined task and are loosely coupled via interfaces to make the system and its components more maintainable, extendable, and portable.

Abstraction: So, what do we do as computer scientists? → we abstract! and we make layers.

Localization: For this we strive for localization, i.e., we define procedures and data types needed for this task, within a local context and encapsulate them there → only limited effect when something changes, more extendable & maintainable.

Black box: To enable modularization a famous strategy is to characterize the components of the DBMS as black boxes. You do not know how the task is implemented exactly but you know what goes in and what goes out. For hierarchical, layered systems this means, that the functionality / information on lower layers is hidden and only an interface is presented which upper levels can use.

Completeness: We want to have a complete description of a DBMS, at least on an abstract level.

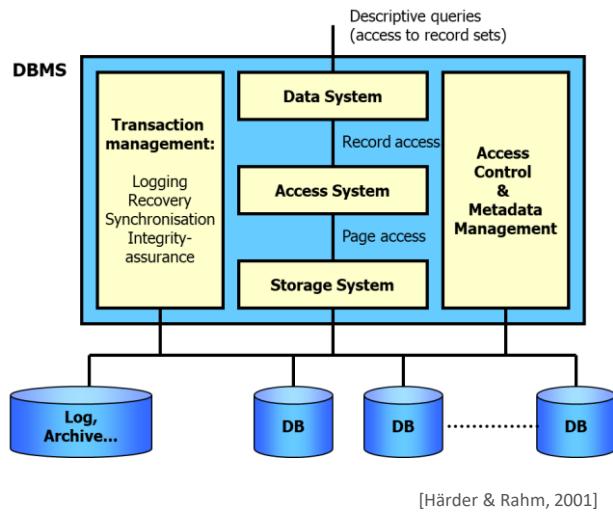
Verifiability: Because we split up the system into independent modules, it is also easier to verify that the system is correct as we can test them independently and make integration tests

Standard methods from software design can be used for the system design, using modular concepts: for example,

- functional abstraction → data and algorithms are handled separately → data has to be handed over or declared globally, not so often in DBMS
- data abstraction → data and procedures are encapsulated together, objects have a state, abstract properties are defined, but not the concrete implementation,
Example: Abstract Data Types
- and object-oriented modularization using objects and their abilities in form of methods, they provide functional abstraction and data abstraction

For DBMS the modularization concept has been used to design a hierarchical layered architecture model as we will see in the following.

Simplified Architecture



[Härder & Rahm, 2001]

16

IMPLEMENTATION OF DATABASES 2023 / 2024

Here we see a simple version of a layered architecture of a DBMS.

The advantage of a hierarchical layered system is, that the layers provide an interface to the layer above them, hiding the lower layers.

Higher layers get simpler, as they can build on the lower levels.

Changes on upper layers do not influence the lower layers.

Example: When we provide an implementation of a join on a lower level, we can just use the join in relational algebra

Assumptions:

- DB is not stored in the main memory of the computer concerned, but in the secondary memory
- Suitable data structures are used for efficient secondary memory management.
- User has descriptive language, with which she can access and manipulate the data

Let us start at the lowest layer, **the storage system**.

The storage system manages the access to the main memory and the external storage (hard disks).

In the main memory it maintains a database buffer where it provides the data

organized in pages.

It mocks a page-oriented main memory database and provides page access to higher levels.

Hence, it hides aspects of mapping to external storage.

The **access system** encapsulates the mapping of access paths, such as indexes, and physical records to pages.

This is a very hard part, as the strategy of doing this has to be very efficient as the gap in access times between main memory and

external storage is very high, which we will see on one of the next slides.

To the data system, the access system provides an interface to the physical objects and internal records.

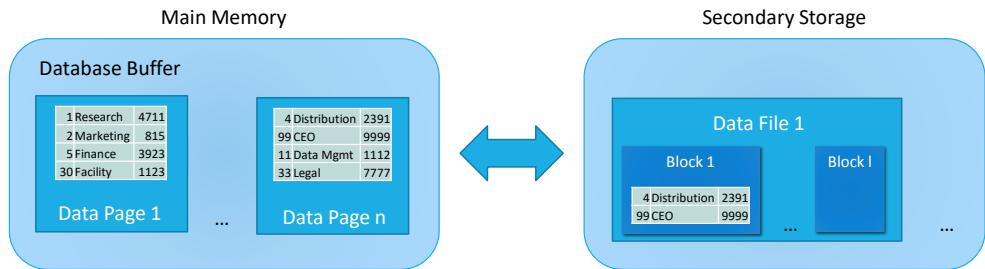
Finally, the **data system** bridges the gap between the set-oriented tuple interface (for relations, views) and the record-oriented processing of the internal records.

I.e., It provides the possibility for users and applications to pose declarative queries to this set-oriented interface.

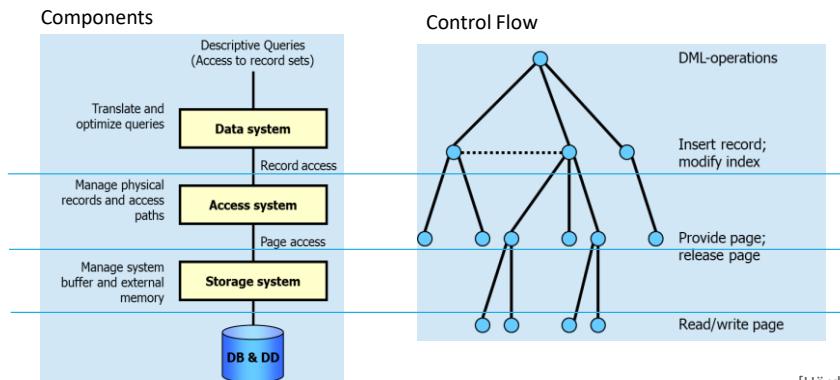
To complete the functionality towards the tasks we defined earlier, we have at the right and the left side service components which take care of transaction management, meta data management, and access control on all levels.

Besides the databases with the actual data in the lower part, these services also require databases to store their maintenance data, such the dictionary or logs.

Data Organization in DBMS



Interaction of the Layers



[Härder & Rahm, 2001]

In this slide the interaction between the layers is depicted. The left diagram shows the components and tasks explained before and the right diagram shows the commands provided by the corresponding interfaces and the control flow graph created by the initial query.

We start at the top of both diagrams.

- A declarative query is sent to the set-oriented interface in form of CRUD (Create, Read, Update, Delete) operations formulated in a DML= Data Manipulation Language (e.g., in SQL)
- The query is then parsed and translated into more formal representations (e.g., relational algebra) and then optimized on different levels in the data system
- The interface of the access system is then used to look up the access paths and records for the operations (i.e., we determine which indexes to use and which records to access).
- The access system also looks up if the corresponding pages are already in the DB buffer.
- If not, it uses the interface of the storage system to load the pages into the buffer (or releases pages)

The set-oriented operations from the start can create very wide control flow graphs
(creation of many operations on lower levels by higher levels)
which offer the opportunity for parallelization of the operations on each level, such
that we can possibly pre-calculate results for example.

As discussed, the gap between RAM and hard disk is a very crucial aspect for the
performance and tuning of a DBMS. We corroborate that on the next slide.

Background Information on Hardware

	Capacity	Speed	Access time
L1-Cache	32-256 KB	1 TB/s	1 ns
L2-Cache	256 KB-4 MB	1 TB/s	4 ns
L3-Cache	> 8 MB	>400 GB/s	~40 ns
RAM	GBs to TBs	>100 GB/s	~80 ns
Solid State Disc	GB – TB	Up to 7000 MB/s	<0.1 ms
Hard disc	TBs	Up to 1.000 MB/s	>3 ms
CD/DVD	640 MB – 20 GBs	10 MB/s	150 ms
Streamer	4 GB - >100 GB	2-10 MB/s	100 ms - >10 s
Network	-	1/10/100/1000 MB/s	ca. 1 ms

→ Prefer main memory, reduce disk accesses!

We mentioned the hard task of the access system to find an efficient strategy for mapping the records and access paths to pages.

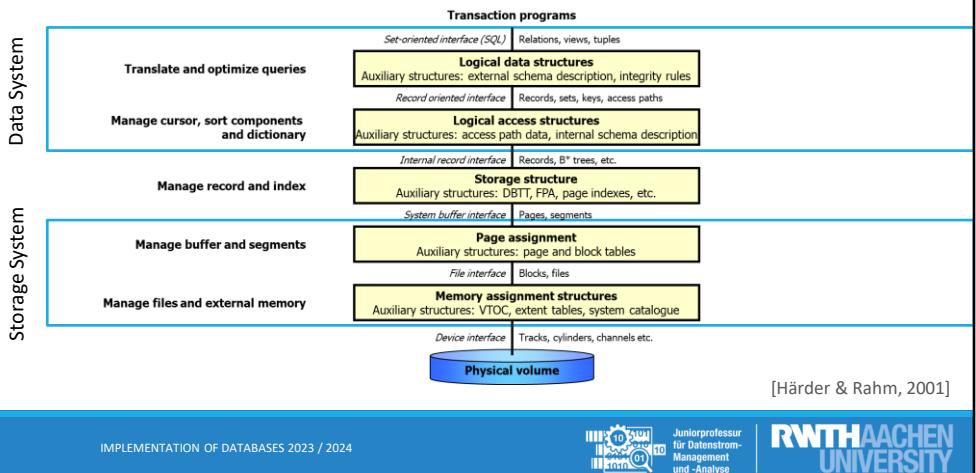
You can see, that the storage close to the CPUs is fast and expensive, but has only limited capacity, while the larger the data volume , the slower the access, but also the cheaper the hardware.

We can see that there is a factor of about 10.000 - 100.000 between RAM and hard disc.

One critical point in performance of a DBMS is hence the big gap of access times between main memory and external storage.

Therefore, the most important point in the optimization of a database system is to do as much as possible in main memory and reduce the number of disc accesses which is the task of the access system.

Five-layer model of a DBS



20

IMPLEMENTATION OF DATABASES 2023 / 2024


**RWTHAACHEN
UNIVERSITY**

The architecture we have seen before can be refined further to a 5-layer model, which explains in more detail the tasks and interfaces of the DBMS components.

In this more detailed architecture, the data system and storage system have been split into two layers.

For each layer we see the interfaces they use and provide and also the objects (files or pages) the interfaces work with.

For example, the Page Assignment layer needs to work with pages and segments as well as with blocks and files.

Set-oriented interface: provides access to logical data structures such as relations, tuples, views and can be accessed using a declarative language such as SQL.

Record oriented interface: provides access to external records, sets, keys, ... : FIND NEXT record, STORE record. It realizes the independence from the storage structures.

In some DBMS (e.g., OODBS) it is used as an API.

The **Logical Data Structure layer** enables the mapping between relation, views and tuples to records, sets and so on.

Internal record interface: provides access to records, b*-trees and other index structures, e.g., with store record, insert entry into b-tree commands.

The corresponding **Logical Access structure** maps records, sets keys and access paths to the corresponding internal records and organizes them in e.g., B*-trees.

System buffer interface: enables access to pages and segments: e.g., get page, (de)allocate page.

The **storage structure layer** organizes records and access paths into pages using DBTT: database key translation table (maintains a mapping between page addresses and internal records), FPA: Free Place Administration (manages free space on the pages), pages indexes and more

File interface: enables access to blocks and files, e.g., read/write block/file.

The **page assignment layer** then organizes blocks and files into pages.

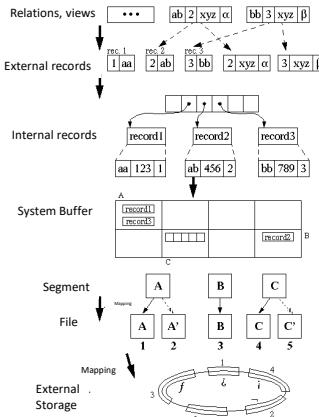
It utilizes page tables (data structure used by virtual memory system to maintain mapping between virtual and physical addresses) and block tables (maintain blocks of virtual memory)

Device interface: is provided by OS or hardware and encapsulates access to tracks, cylinders etc. on the hardware.

The **memory assignment layer** organizes the memory structures, such as tracks or cylinders into files and blocks.

VTOC: Volume Table of Contents (maintains table of contents of a volume and their physical location on the disk and some metadata)

Example



```

SELECT p.Name FROM Professor p, Vorlesung v
WHERE   p.Rang='C4' AND v.Titel='Logik'
        AND p.PersNr = v.gelesenVon

{p.name | p∈Professor ∧ ∃v∈Vorlesung ∧ p.Rang=C4
        ∧ v.Titel=Logik ∧ p.PersNr=v.gelesenVon}

ΠName(σTitel=Logik ∧ Rang=C4(Professor) πPersNr=gelesenVon Vorlesung)

ΠName(σRang=C4(Professor) πPersNr=gelesenVon
        σTitel=Logik(Vorlesung))

OPEN CURSOR Vorlesung(Titel='Logik')
FIND NEXT record ...
OPEN CURSOR Professor(Rang='C4')
...

B←TREE-SEARCH Vorlesung(Titel='Logik')
FETCH RECORD Vorlesung(...gelesenVon)
...
B←TREE-SEARCH Professor(PersNr=gelesenVon)
...

LOAD PAGE 123
WRITE PAGE 345
...
  
```

[Härder & Rahm, 2001]

Data Independence: An Overview

Layer	What is hidden?
Logical data structures	Position indicator and explicit relations in the schema
Logical access paths	Number and kind of the physical access paths; internal representation of records
Storage structures	Management of buffers, logging
Page assignment structures	File mapping, indirect page assignment
Memory assignment structures	Technical features and technical details of external storage media

Problems:

Due to high specialization, functionality of operating system often not usable

- Segment-file mapping
- Paging
- Shadow memory
- Buffer management
- Dispatching

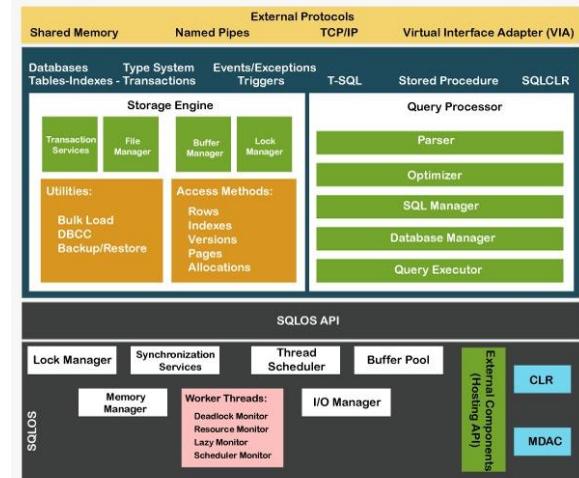
- Logical data structures: obscure position indicator (points to currently viewed tuple) and explicit relations in the model
- Logical access paths: Hide number and type of physical access paths; internal representation of data sets
- Memory structures: Obscure buffer management and logging
- Page allocation structures: Hide file mapping, indirect page allocation.
- Memory allocation structures: Hide technical properties and technical details of external storage media

To realize these features, algorithms for paging, dispatching and so on are needed, which are integrated into the OS

But due to high specialization, functions of the operating system cannot be used

Cheap DBMS use many functions of the OS, while expensive DBMS implement functionality themselves that could also be provided by the OS.

Example: MS SQL Server Architecture



<https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-guides?view=sql-server-ver15>

IMPLEMENTATION OF DATABASES 2023 / 2024

SQLoS provides user-level operating system services (hence the name “SQLoS”) to the rest of the server. Components in the SQL Server Engine make use of the services provided by SQLoS to schedule individual or multiple tasks, allocate memory, and so forth.

The key observation here is that DBMS and OS schedulers must work together. Either OS must have built-in support for DBMS or DBMS must have a special scheduling layer.

The SQLoS behaves very much like an operating system. It abstracts the concept of memory management, I/O, scheduling, etc. from the other components within the SQL engine. In this way, these components do not need to worry about managing things like NUMA and Resource Governor, they simply make resource allocation calls to the SQLoS via an API.

CLR: Common Language Runtime, executes managed code

MDAC: Microsoft Data Access Components



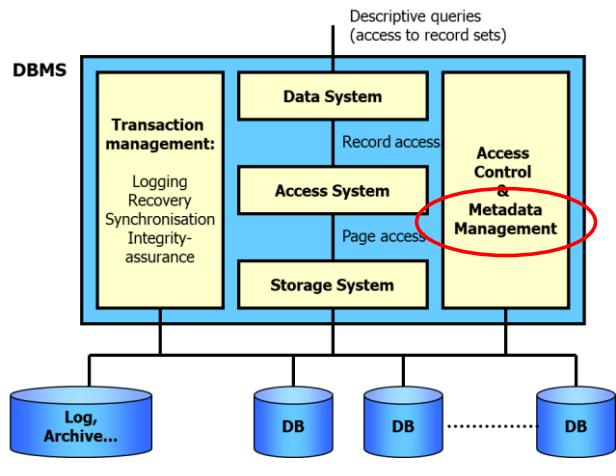
Quiz

<https://www.menti.com>
Code **3951 6372**



24

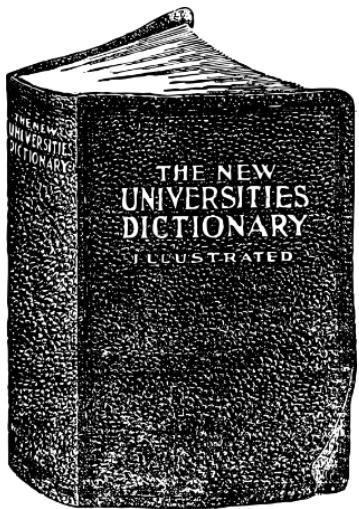
Simplified Architecture



[Härder & Rahm, 2001]

Implementation of Databases 2023 / 2024

25



DBMS with Data Dictionary

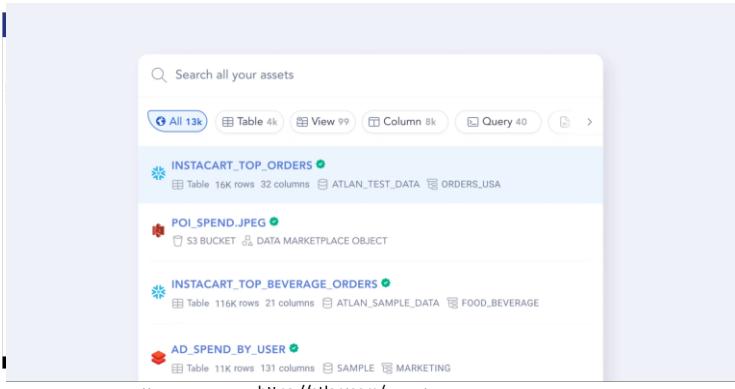
- Problem: each abstraction step means loss of semantics
- Realization of operations requires information about
 - Schemas
 - Integrity constraints
 - Index structures
 - Access authorization
 - ...
- Approach: Comprehensive management by data dictionary
 - Internal in DB (uniform model)
 - Stand-alone module (fast and specialized service)

Example: Internal Data Catalogue

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the database structure under 'Servers (1)'. Under 'PostgreSQL 13', there are 'Databases (2)' containing 'postgres' and 'university', and 'Catalogs (2)' containing 'information_schema' and 'pg_catalog'. The 'pg_catalog' node is selected. The main pane shows the 'Query Editor' with the query 'SELECT * FROM pg_database;'. Below the query editor is a table titled 'Data Output' showing the results of the query:

oid	datname	oid	encoding	datacollate	datctype	datistemplate	datislink
1	13442	postgres	10	English_United States.1252	English_United States.1252	false	true
2	16394	university	10	English_United States.1252	English_United States.1252	false	true
3	1	template1	10	English_United States.1252	English_United States.1252	true	true
4	13441	template0	10	English_United States.1252	English_United States.1252	true	false

Example: External Data Catalogues



The screenshot shows the Microsoft Azure Data Catalog interface. At the top, there is a search bar labeled "Search all your assets". Below the search bar, there are several filters: "All 13k", "Table 4k", "View 99", "Column 8k", "Query 40", and a refresh button. The main area displays a list of registered data assets:

- INSTACART_TOP_ORDERS**: Table, 16K rows, 32 columns, located in ATLAN_TEST_DATA, under ORDERS_USA.
- POI_SPEND.JPG**: S3 BUCKET, DATA MARKETPLACE OBJECT.
- INSTACART_TOP_BEVERAGE_ORDERS**: Table, 116K rows, 21 columns, located in ATLAN_SAMPLE_DATA, under FOOD_BEVERAGE.
- AD_SPEND_BY_USER**: Table, 11K rows, 131 columns, located in SAMPLE, under MARKETING.

At the bottom of the catalog interface, there is a URL: <https://docs.microsoft.com/en-us/azure/data-catalog/register-data-assets-tutorial>.

Below the catalog interface, there is a blue footer bar with the following information:

- 28
- IMPLEMENTATION OF DATABASES 2023 / 2024
- 
- Juniorprofessur
für Datenstrom-
Management
und -Analyse
- RWTH AACHEN**
UNIVERSITY

The Azure Data Catalog is a cloud-based metadata repository in which data sources and data assets can be registered.

The catalog serves as a central storage location for structural metadata extracted from data sources and for descriptive metadata added by users.

Other examples are: Alation Data Catalog, Atlan Enterprise Data Catalog, Talend Data Catalog, Collibra Data Catalog, Informatica Enterprise Data Catalog, Microsoft Azure Data Catalog, Oracle Cloud Infrastructure Data Catalog, Google Data Catalog

Anzo Cambridge Semantics is one example of a system utilizing vocabularies and is built upon the open data standards OWL, RDF and SPARQL.



1.3 Evolution of DBMS

Learning Goals

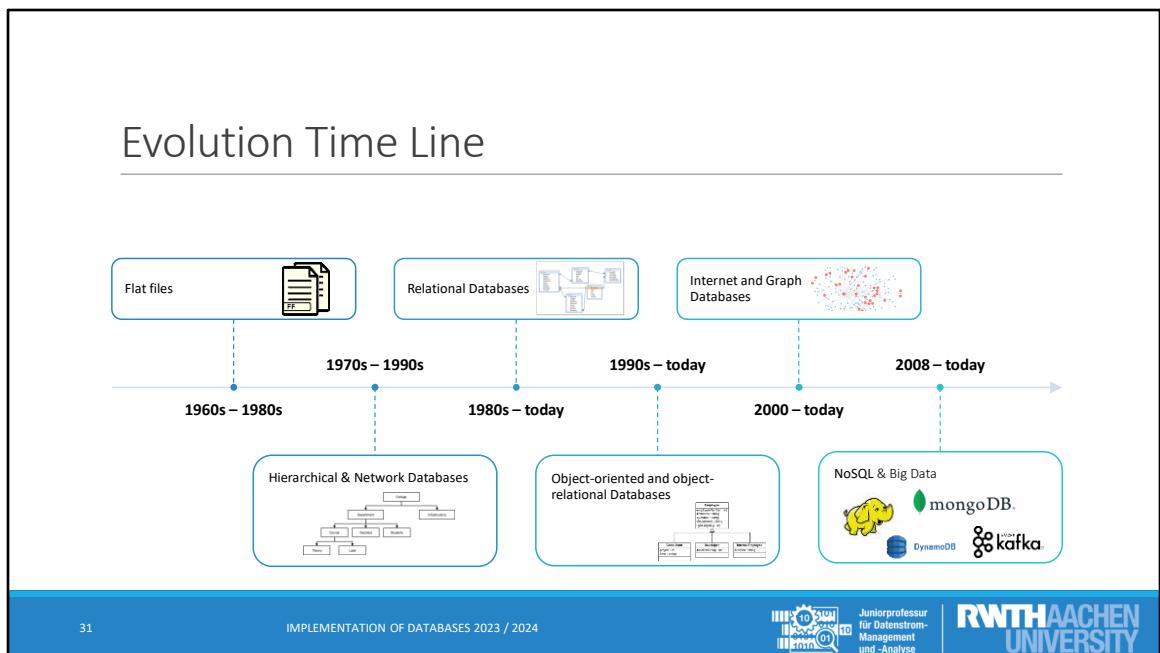
At the end of this section, you will be able to

- ✓ name the most important stages in DBMS evolution
- ✓ classify a DBMS along different dimensions
- ✓ name different client-server architectures and discuss pros and cons
- ✓ explain tasks of client and server and name three different server types
- ✓ explain distributed database principles
- ✓ explain what is a big data architecture

30



Evolution Time Line



Flat file database

- Database stored in a single file / table
- No explicit relationships, uniform format, no constraint checks
- Example: dBase, SQLite, Berkeley DB, CSV

Hierarchical & network databases

- Basis of pre-relational systems
- Data is conceptually described with the help of trees (hierarchical) or graphs
- nodes = record types, edges = (binary) relations between record types
- Database = collection of files linked in a special way
- Problems: intermixing of conceptual relationships with the physical storage and placement of records on disk → no Data Independence
- Very complex application programs were created due to cumbersome access language → sometimes for new queries everything had to be reimplemented
- Only programming language, no higher-level user interfaces / languages
- Executed on large expensive mainframe computers

Relational:

- proposed to separate the physical storage of data from its conceptual representation
- provide a mathematical foundation for data representation and querying
- introduced high-level query languages can be used instead of programming language → can write new queries fast
- Separates data and application → improved DI
- First systems quite slow, but then indexing and storage structures evolved, query optimization

Object-oriented, object-relational

- emergence of object-oriented programming languages in 1980s
- Need to store and share complex, structured objects
- Incorporate many oo-paradigms
- But: Complex, no standards → only limited usage, no impact in the market
- Used for engineering applications, manufacturing
- Incorporated into relational systems → ORDBMS

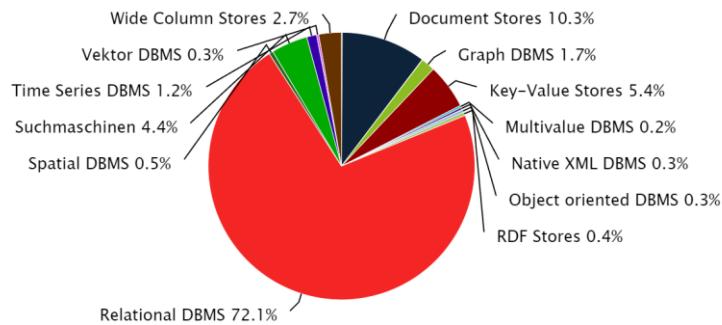
Internet and Graph Databases

- Internet -> large amount of interconnected computers
- Link documents via hyperlinks (pointer)
- In late 1990s XML as standard to exchange data over the web
- RDF based on XML, graphs, enabling complex network analysis, semantics, with OWL also derivation of new knowledge

NoSQL

- High variety of data available
- Performance got a major aspect , due to high loads in the internet (web shops, social media etc.), 24/7 availability
- Lot of simple and uniform queries and updates → need simple, but efficient data models
- Traditional dbms → not flexible enough, not scalable enough, not elastic, not fast enough
- NoSQL → problems with ACID, e.g., not normalized models

Data Model Popularity Today



© 2023, DB-Engines.com

Db engines uses various criteria to estimate the popularity, e.g., appearance on websites using search engine hits, number of searches, technical discussions in Stack Overflow, job offerings, social networks

Classification Dimensions of DBMS [Elmasri & Navathe, 2017]

Data Model

Number of users

Number of sites

Costs & License

Types of access path

Purpose

In principle we can classify systems along various dimensions.

Data Model: we have seen this in the last slides, it influences the implementation of DBMS

Number of users: Single user systems vs. Multi-user systems (have to take special care)

Number of sites the system is distributed. Centralized (single site) vs. Distributed DBMS over many sites connected via network, homogenous vs. Heterogeneous, client-server vs. Federated (local autonomy)

Cost & License: open-source, many commercial systems, based on accessing users / installations, on-demand / usage (cloud)

Types of access path options for storing files, e.g. inverted file structures.

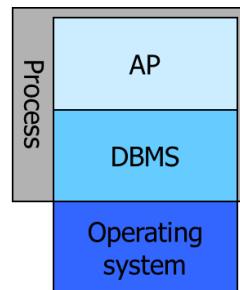
Purpose: special purpose for certain applications vs. general-purpose (e.g., airline reservation systems, telephone directory systems) → cannot be used in other

contexts

Embedded DBMS: 1-Layer-Architecture, Single User DBMS

- One process, one address space
 - No concurrency control
 - Simple crash recovery
-
- AP: Application program with DB calls
 - DBMS: Belongs to the address space of the AP

Example: PC database systems (MS Access)



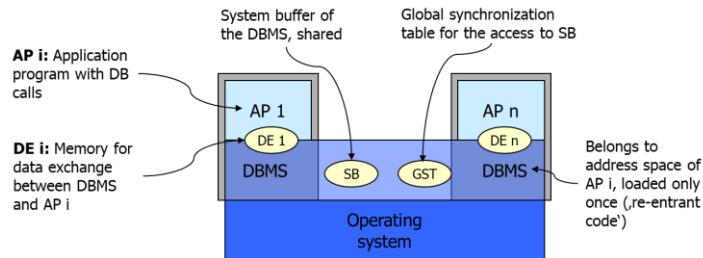
For 1-Layer Architecture → is a centralized DBMS, runs all functionality on one computer

DBMS is part of the address space of the application program

Limitations, not secure, not usable by several users

Embedded DBMS: 1-Layer-Architecture – Multi-user DBMS

- Multiple processes, communication via shared address space
- Very efficient data exchange via shared memory, but
- No security concerning errors in the AP



Example: IBM System/R* – research prototype

As databases become more complex and big early prototypes of DBS evolved to Multi-user DBMS with multiple processes and communication over shared access space in main memory, efficient, but no security in terms of errors of the application program

Multiple processes → DBMS is loaded only once, however, since reentrant code is used
→ only one copy of the code

Synchronization table (global synchronization table) required → if one process changes the shared address range, the other process is not allowed to access it for the time being
A process locks objects in the address range for editing in the synchronization table → inefficient

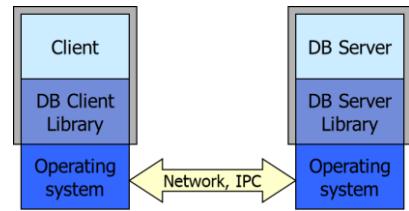
There is also a buffer for data exchange between the application and DBMS, from which data can be loaded into or unloaded from the shared buffer.

Ex: IBM System/R* - research prototype

Problems: very difficult to implement, slow, and OS needs to take care of the protection of the shared buffer → insecure, since not properly protected

2-Layer-Architecture (“Client/Server”)

- Client and server totally separated \Rightarrow distributed access to DB
- Communication among clients and servers via a network or IPC.
- Specialized protocols used (JDBC, Net8, TCP/IP)
- Clear separation of client and server



Client and server totally separated \rightarrow distributed access to DB

Communication between clients and server via network or IPC
(interprocess communication)

- There are special communication protocols (e.g., JDBC, ODBC, Net8, TCP/IP)
- Clear separation between client and server
- Client and server do not have to have the same OS

Client:

- AP client + DB client library
- writes data in server and reads it out
- Client does not need the whole functionality of the DBS, but only the user interface and local buffer

Server:

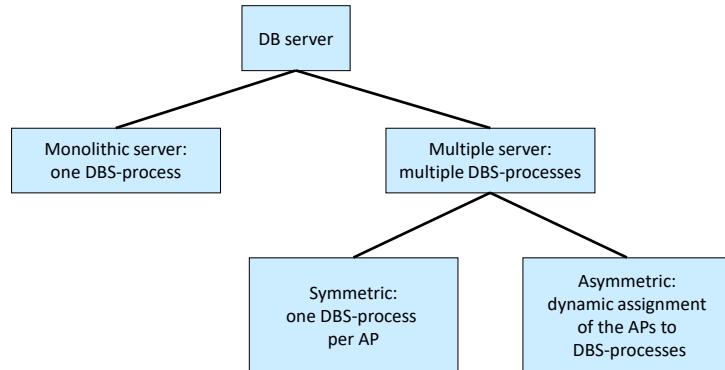
- DB server + DB server library
- does the main work \rightarrow mainly manages the data and “takes the role of shared

buffer and GST."

- does not need the user interface, only needs the storage functionality and the synchronization functionality, Recovery
- * can be very big with multiple processors and large memory arrays

The client-server architecture can be categorized /refined in more detail as follows.

Classification Client/Server Architecture



Categorized along how many processes we have for the server and if there are multiple server processes how they are assigned to application processes.

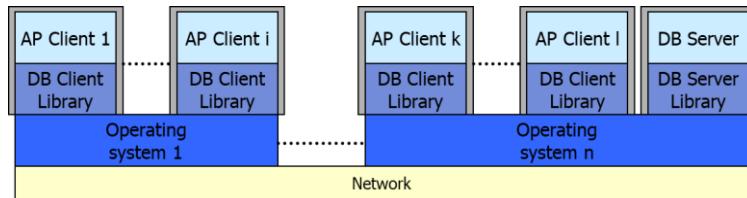
Single DB Server Process

- Synchronized access to system buffer and central system tables
- Server uses multi-threading ("re-entrant code")
- Only one server process for many clients
- DB server process is preferred by OS



Monolithic Server

- Own resource management, duplicates OS functions
- Simple communication in the server via shared memory
- Example: PostgreSQL



- Own resource management, duplicates OS functions and hides those in OS
- Easy communication in the server via shared memory
- Clients do not have peer-to-peer connection → cannot communicate with each other → must communicate via shared memory in server
- For medium sized databases

Multiple DB Server Processes

- Communication among the servers via “shared memory”
- Communication among clients and servers/dispatcher via OS-mechanisms (IPC) or network software
- **Symmetric assignment**
 - Each client is assigned to exactly one server process
 - Static assignment, fixed number n of servers stated in advance
→ maximal degree of parallelism is n
- **Asymmetric assignment**
 - Each client is assigned to a server process by a dispatcher
 - Fixed number n of servers stated in advance, but degree of parallelism can be higher.



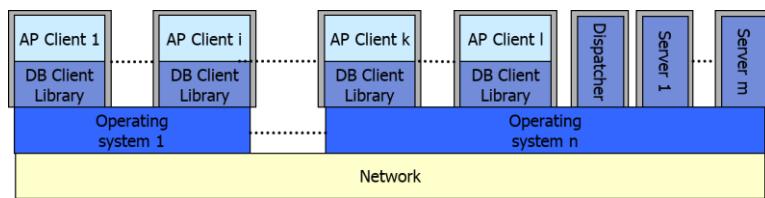
For very large databases and many users, where parallelism is required

Asymmetric

- dynamic allocation of applications to DBS processes → can be used as needed, e.g., a lot of processor power at certain times of the day to an application that is then busy, allocate
- Can extend this to a 3-tier architecture where, application, server, and client are completely separate

Multiple DB Server Processes - 2

- DBMS is a compound of different processes
- Communication via operating system or network
- Process scheduling by OS, advantageous in multi-processor computers, because OS manages processor allocation.
- Example: Oracle, IBM DB2, MySQL, MS SQL Server, Sybase



Symmetric assignment: $l \leq m$

- DBMS is composition of various processes communication through the OS (on one computer) or network
- Process scheduling by the OS, advantageous for in multiprocessor computers. → OS manages processor allocation
- Oracle (client-server): software based only, OS and hardware independent.
- IBM DB2 (mainframe): no client-server, multiple servers, but in a supercomputer with many processors and disks (mainframe), dispatcher has more control over all servers, hardware based
- Dispatcher (on other server): establishes connection between clients and the right servers

Tasks of Client and Server

- Tasks of a server

- Data management
 - Relation Server
 - Object Server
 - Page Server
- Application functionality



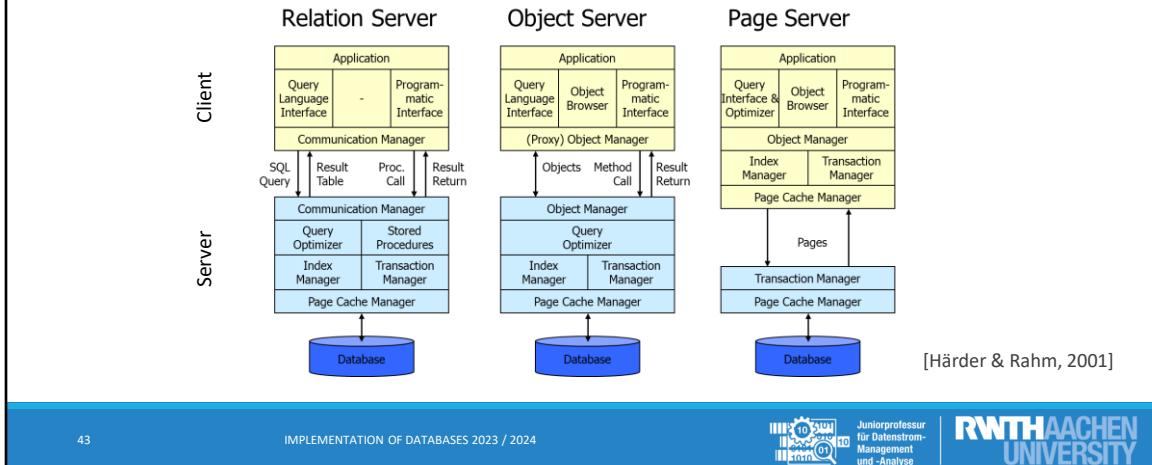
- Tasks of a client

- Presentation of data
- Application functionality

- Differences of client and server in a client/server architecture
- Server manages data and may also have application functionality
- Client presents data and may also have application functionality.

Depending on how much functionality is embedded in a client, we talk either of a thick or a thin client.

Server Types



43

IMPLEMENTATION OF DATABASES 2023 / 2024



RWTH AACHEN
UNIVERSITY

Relation Server

- Classic, relational, distributed database
- Most widely used
- SQL queries between client and server via communication manager
- All functionality of DBMS at server
- Need cursor management at the client → enables element-by-element processing of the data records
- Relatively thin client, but application can be large

Advantages

- Distribution of data to clients would greatly limit efficiency due to high communication overhead between clients and server required
- SQL expression is processed at server → bears the high with complex optimization → client would only be able to do this if it could client would only be able to do this if it took over a lot of database functionality (mostly clients are not capable of doing this)
- Locality of data can be kept transparent

Object Server

- Only objects are exchanged via object manager (proxy) → Application objects are provided by the server.
- Application objects are provided by the server
- Server is responsible for a large part of the database-specific tasks
- Typical for OO databases
- Client requests objects based on their object identity
- High memory requirements and implementation effort
- Ex: CORBA → Exploitation of object services

Page Server

- Make client fat → client knows SQL and index management and does much of transaction management
- "dumb server" → mainly manages secondary storage
- Client does main work of converting requests and updates
- Pages are exchanged
- Client requests from server only certain pages at their address
- Only the lower layers (buffer management, etc.) are handled by the server takes over
- Used when you do not have a single data model, e.g., XML and ER etc. at the same time
- Server knows nothing about logical structure of DB
- Used a lot also in OODBMS: EXODUS, OS, ObjectStore...



Quiz

<https://www.menti.com>
Code **3951 6372**



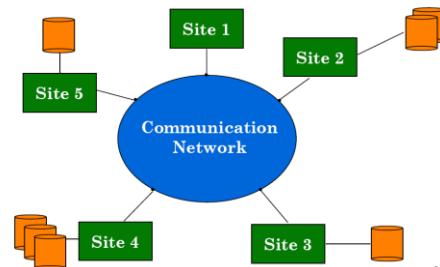
44

Distributed Database Systems

- A distributed database (DDB) is a collection of multiple, *logically interrelated* databases distributed over a *computer network*
- A distributed database management system (D–DBMS) is the software system that permits the management of the distributed DBs and makes the distribution *transparent* to the users

Distributed database system (DDBS)

$$\text{DDBS} = \text{DDB} + \text{D–DBMS}$$



[Özsu & Valduriez, 2011]

- Multiple computing nodes connected via a network, where each node contains an instance of the DBMS and a part of the database
- Example
 - branches of a bank
 - Each branch maintains the accounts of its local customers
 - But the head office also needs to access all of them

Promises of D-DBMS

- **Transparent** management of distributed, fragmented, and replicated data
- Improved **reliability/availability** through distributed transactions
- Improved **performance**
- Easier and more economical system expansion → **Scalability**



- Partitioning: Fragmentation and allocation of data
- Replication

Implicit Assumptions



Data stored at a number of sites

Each site *logically consists of a single processor.*



Processors at different sites are interconnected by a computer network

no multi-processors → parallel database systems



Distributed database is a database, not a collection of files

Data is logically related as exhibited in the users' access patterns
→ relational data model



D-DBMS is a full-fledged DBMS

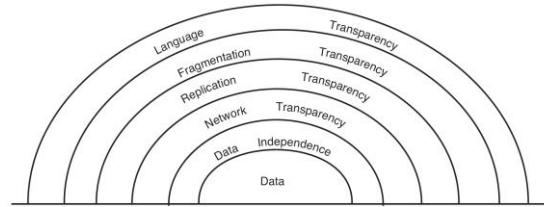
→ not remote file system, not a Transaction Processing system

Transparency

... is the separation of higher-level semantics of a system from lower-level implementation issues

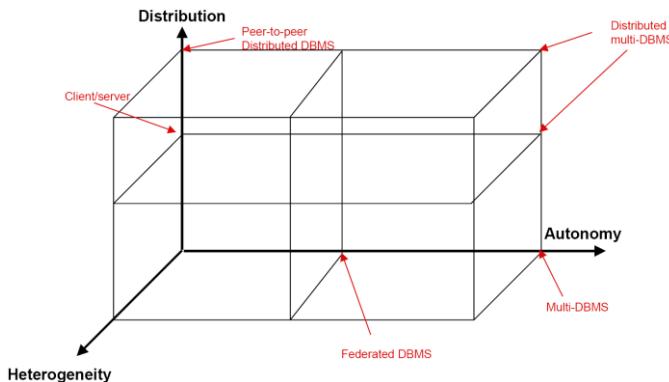
.. has the goal to provide **data independence** in the distributed environment

- Network (distribution) transparency
- Replication transparency
- Fragmentation transparency
 - horizontal fragmentation: selection
 - vertical fragmentation: projection
 - hybrid



- Network transparency: users are unaware, that the system they access is distributed over a network
- Replication transparency: user is unaware that the data is replicated
- Fragmentation: do not have to know where the fragments are and what they are
→ fragmentation is transparent
- Language: the users need not be concerned if different servers support different query languages

D-DBMS: Implementation Alternatives



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are –

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.
- Types of Homogeneous Distributed Database

Autonomous – Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.

Non-autonomous – Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are –

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- **Federated** – The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Multi-DBMS** – The database systems employ a central coordinating module through which the databases are accessed, but they are in principle working autonomously

Big Data Architectures

- Big Data requires large, scalable, distributed architectures
- Four/Six Vs : Volume, Velocity, Variety, Veracity, [Hofstee & Nowka, 2013], (Value, Variability)
- Heterogeneity
 - Sources
 - Systems
 - Requirements
 - Client Applications

Big Data Systems are *complex ecosystems*, independent components have to be integrated

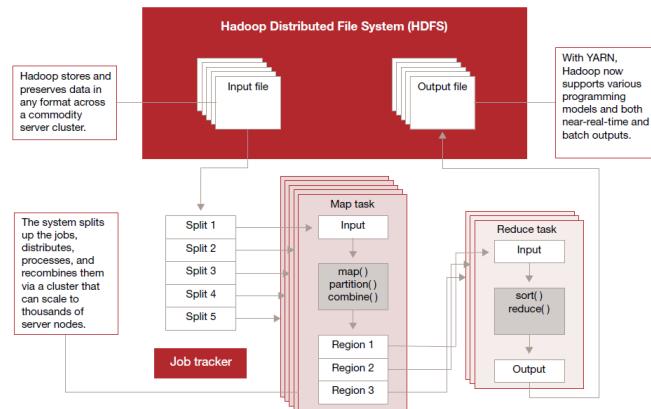
Hadoop is not a Big Data system, it is just a component!

Variability: Variance in the data, such as changes due to seasonality (concept shift)

Value: value of the data → do we need the data?

Hadoop as a Basic Big Data Platform

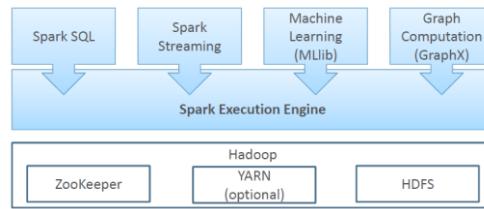
A basic Hadoop architecture for scalable data lake infrastructure



Source: Electronic Design, 2012, and Hortonworks, 2014

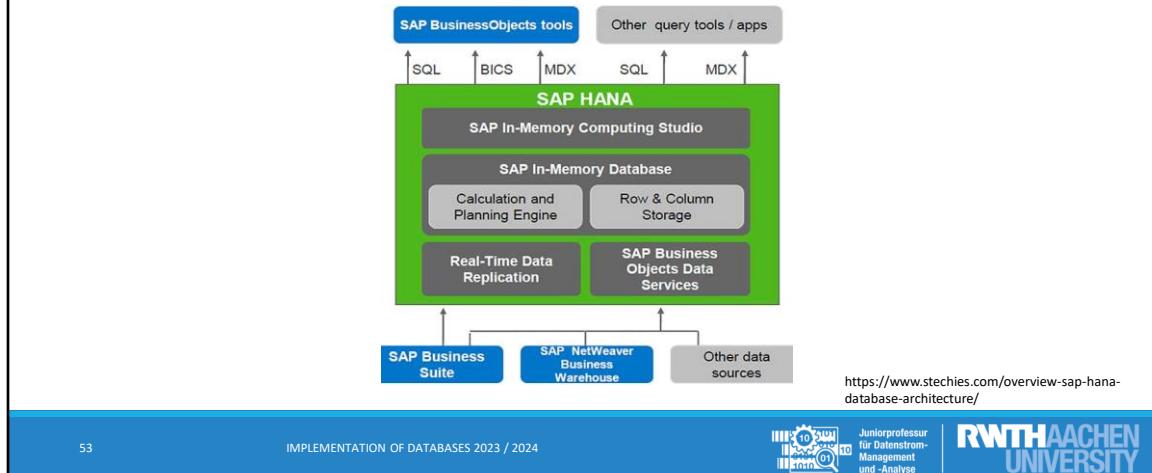
In-Memory Database Systems

- Distributed data processing, but do as much as possible in-memory and avoid I/O operations (to disk or distributed file system)
- Example: Apache Spark
 - Distributed In-Memory Computing Framework
 - General framework for all kinds of SQL and non-SQL analytics



[Albrecht, 2015]

Another Example for an In-Memory Database System



53

IMPLEMENTATION OF DATABASES 2023 / 2024



Juniorprofessor
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY

SAP HANA is a column-oriented, in-memory database, that combines OLAP and OLTP operations into a single system; thus, in general SAP HANA is an online transaction and analytical processing (OLTAP) system, also known as a hybrid transactional/analytical processing (HTAP).

Very costly → high performance, but usually most of the data is not really used... Only for subsets of data to be stored which are really accessed and used frequently.

Quiz

<https://www.menti.com>
Code **1692 3316**



54



References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

- [Albrecht, 2015] J. Albrecht: Processing Big Data with SQL on Hadoop. TDWI, 2015.
- [Boci & Thistletonwaite, 2015] Boci, E. & Thistletonwaite, S.: A novel big data architecture in support of ADS-B data analytic *Proc. Integrated Communication, Navigation, and Surveillance Conference (ICNS)*, 2015, C1-1-C1-8
- [Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). *Fundamentals of database systems* (Vol. 7). Pearson.
- [Härder und Rahm, 2001] Härder, T. & Rahm, E. Datenbanksysteme: Konzepte und Techniken der Implementierung Springer Heidelberg, 2001 (in German)
- [Hofstee & Nowka, 2013] Hofstee, P. and Nowka, K. J. (2013). The Big Deal about Big Data - A Perspective from IBM Research. Presentation at IEEE NAS Conference, Xi'An China.
- [IRDS Framework Standard, ISO 10027:1990] <https://www.iso.org/obp/ui/#iso:std:iso-iec:10027:ed-1:v1:en>
- [Kemper & Eickler, 2015] Kemper, A., & Eickler, A. (2015). Datenbanksysteme; 10., akt. u. erw. Aufl. (in German)
- [Özsu & Valduriez, 2011] Özsu, M. T. & Valduriez, P. *Principles of distributed database systems* Springer, 2011

Implementation of Databases

Chapter 2: Query Languages and Implementation
of Relational Operators

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

Excercises

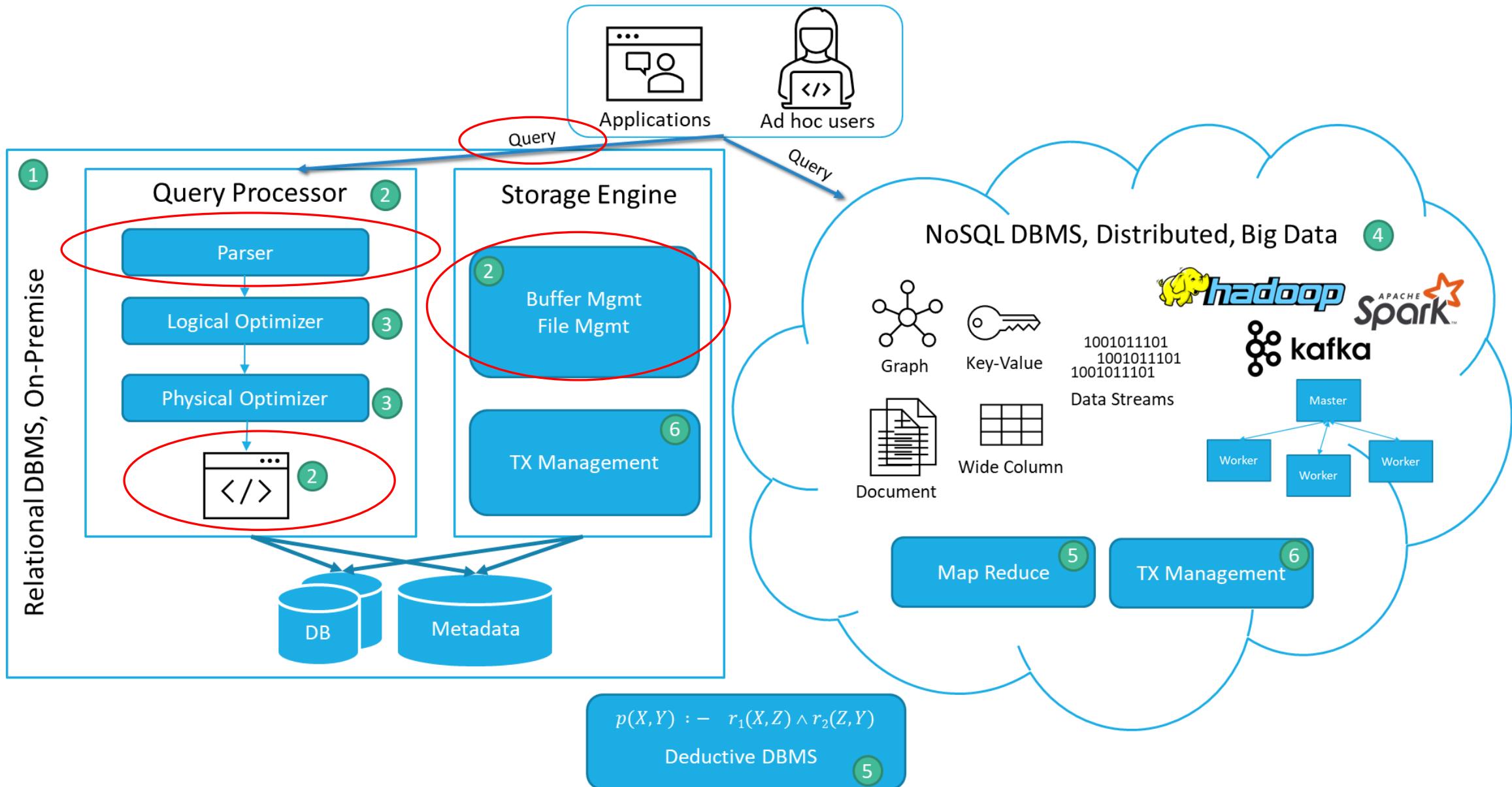
Anastasiia Belova, M.Sc.

Soo-Yon Kim, M.Sc.



Juniorprofessur
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY



Running Example – E-commerce Company Amafon

DB Schema

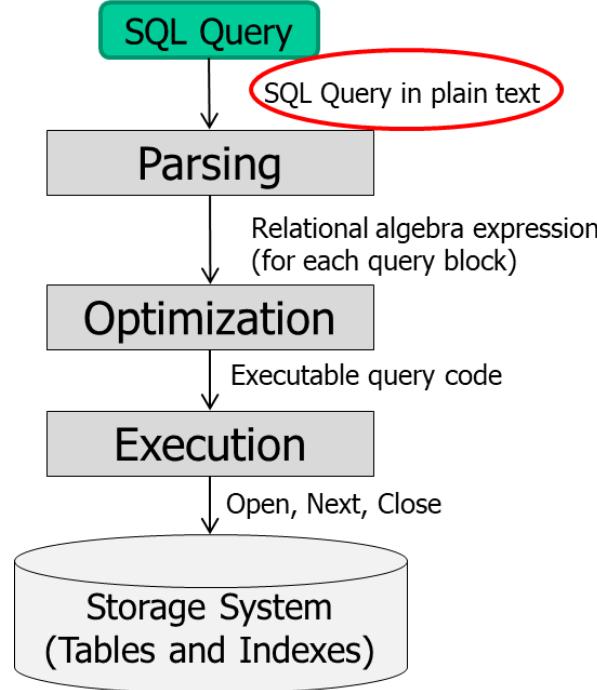
DEPT	EMPL	OFFICE
<u>dno</u>	<u>eno</u>	<u>floor</u>
dname	name	<u>room</u>
mgr	marstat	<u>eno</u>
	salary	
	dno	

Integrity constraints

- Inter-relational dependencies (**foreign keys**)
$$\text{EMPL[dno]} \subseteq \text{DEPT[dno]}$$
$$\text{OFFICE[eno]} \subseteq \text{EMPL[eno]}$$
$$\text{DEPT[mgr]} \subseteq \text{EMPL[eno]}$$
- Functional dependencies
$$\text{EMPL: } \{ \text{eno} \} \rightarrow \{ \text{name, marstat, salary, dno} \}$$
$$\text{DEPT: } \{ \text{dno} \} \rightarrow \{ \text{dname, mgr} \}$$
- Value constraints
$$\text{EMPL: } 10,000 < \text{salary} < 100,000$$

Query Processing Chain (1)

Query: Names of single employees in Anexa department who make less than 40,000



SQL (declarative)

SELECT

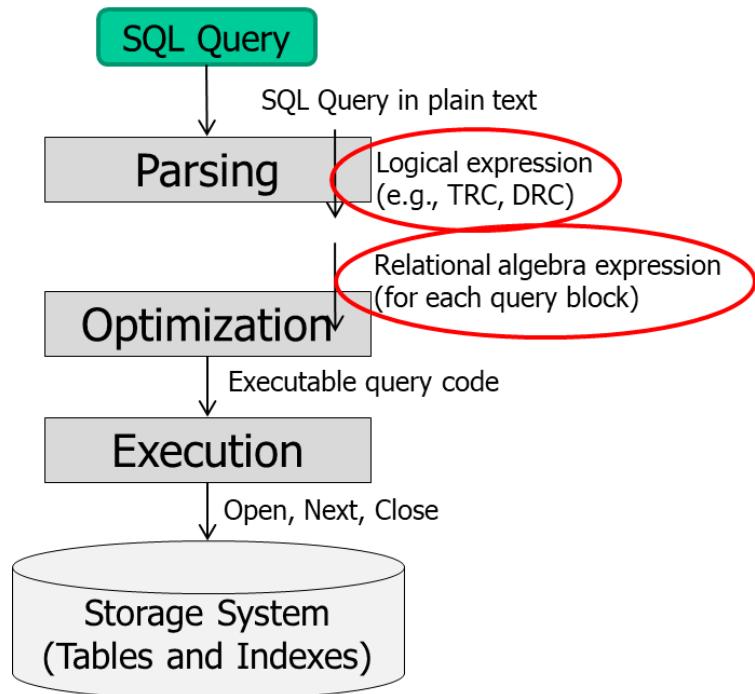
```
e.name FROM EMPL e, DEPT d
```

WHERE

```
e.salary < 40,000 AND  
e.marstat = 'single' AND  
d.dname = 'Anexa' AND  
d.dno = e.dno
```

Query Processing Chain (2)

Query: Names of single employees in Anexa department who make less than 40,000



Tuple Relational Calculus (TRC, declarative)

$$\{ e.name \mid e \in \text{EMPL} \wedge \exists d \in \text{DEPT} \wedge e.salary < 40,000 \wedge e.marstat = \text{'single'} \wedge d.dname = \text{'Anexa'} \wedge d.dno = e.dno \}$$

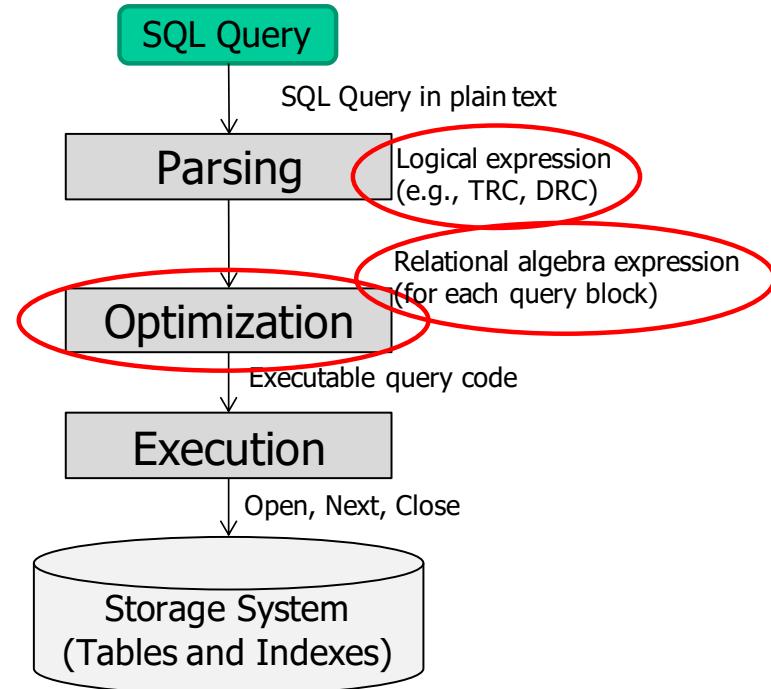
Relational Algebra (RA, procedural)

$$\pi_{\text{name}}(\sigma_{\text{dname} = \text{'Anexa'} \wedge \text{salary} < 40,000 \wedge \text{marstat} = \text{'single'}}(\text{EMPL} \bowtie \text{DEPT}))$$

$$\pi_{\text{name}}(\sigma_{\text{dname} = \text{'Anexa'} \wedge \text{salary} < 40,000 \wedge \text{marstat} = \text{'single'}}(\text{EMPL} \bowtie \text{DEPT}))$$

Query Processing Chain (3)

Query: Names of single employees in Anexa department who make less than 40,000



Relational Algebra (optimized)

$$\pi_{\text{name}}(\sigma_{\text{salary} < 40,000 \wedge \text{marstat} = \text{'single'}}(\text{EMPL}) \bowtie \sigma_{\text{dname} = \text{'Anexa'}}(\text{DEPT}))$$

Query Evaluation and Optimization



Large number of optimization strategies possible. Reduction by
Syntactic query transformation
Semantic query transformation



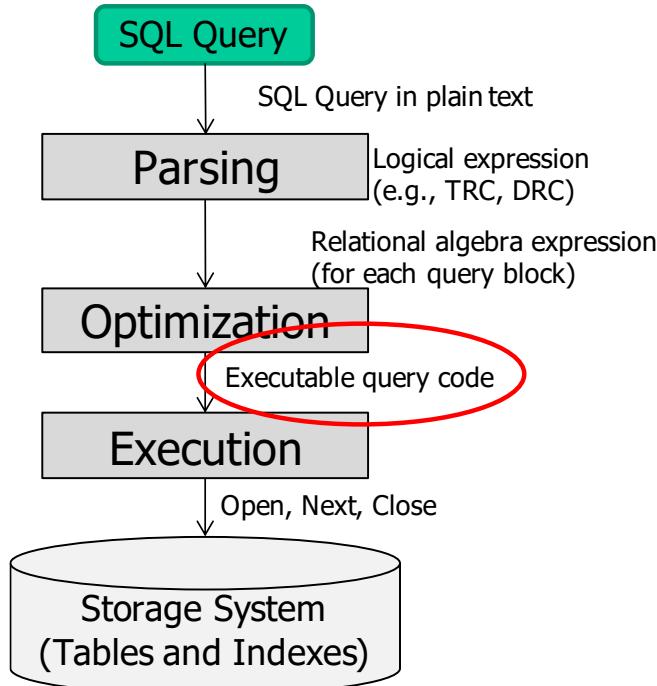
(Implicit) enumeration and evaluation of remaining strategies
Definition of the space needed for data structures and operations
Effects of operations on the size of intermediate results
Effects of the size of intermediate results and operations on communication costs, storage costs and CPU costs



Query support and query optimization by “investments”:
Sorting
Index access
Access paths
Partial operation evaluation

Query Processing Chain (4)

Query: Names of single employees in computer department who make less than 40,000



Nested Loop Join

```
ANSWER:=[];
FOR EACH e IN EMPL DO
  FOR EACH d IN DEPT DO
    IF e.salary<40,000 AND
      e.marstat='single' AND
      d.dname='Anexa' AND
      d.dno=e.dno
    THEN ANSWER:=+ [<e.name>];
```

- Iterate over both relations using two nested loops
- Check join and selection conditions in inner loop
- If ok, add projected attribute(s) to result set

Improved Nested Loop Join

Heuristics to improve query execution: Selection before join!



Improved Nested Loop Join

1. Scan one relation, check selection conditions, put result into temporary buffer
2. Scan second relation, check join & selection condition using intermediate result from temporary buffer, create result set

```
DNOLIST:=[];
FOR EACH d IN DEPT DO
    IF d.dname='Anexa' THEN DNOLIST:+[<d.dno>];
ANSWER:=[];
FOR EACH e in EMPL DO
    IF e.salary < 40,000 AND e.marstat='single'
        THEN FOR EACH d IN DNOLIST DO
            IF d.dno=e.dno THEN ANSWER:+[<e.name>];
```

Summary

Queries can be represented in various languages

Query evaluation requires transformation of queries from a user-friendly query-language (e.g., SQL) to an implementation-oriented language (e.g., RA)

Query optimization can be applied along this transformation process (e.g., syntactical and semantical optimizations), but the choice of operators and access plans is significant for the query performance

Kind of operator implementation also can make a difference



2.1 Query Languages in a Nutshell

Learning Goals

At the end of this section, you will be able to

- ✓ understand and define SQL queries
- ✓ understand and define RA, TRC & DRC expressions
- ✓ discuss complexity and expressiveness of queries



Requirements for Query Representation

Usability

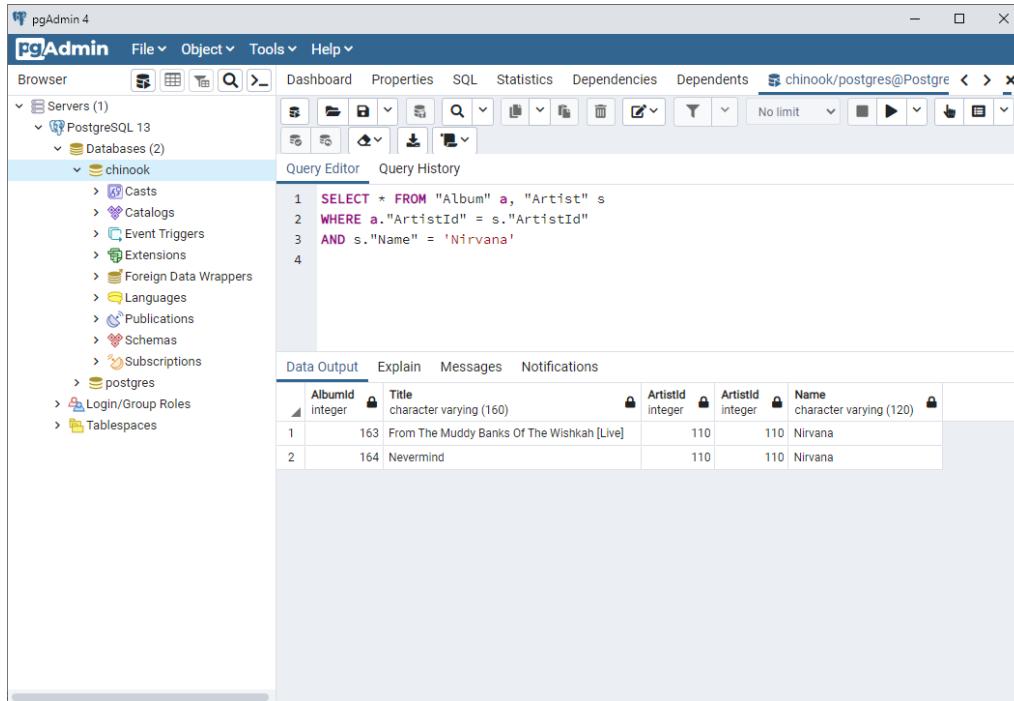
The representation should be appealing and comprehensible for the user

Expressiveness vs. Complexity

Formulation of desired queries should be possible (Standard: the “Relational Completeness”)

Formal manipulability,
commutability,
parallelism, and
pipelining

2.1.1 SQL



- SQL = Structured Query Language
- Standard language for most relational DBMS
 - standardized since 1986
 - current version ISO/IEC 9075:2023, with new supporting JSON data type, property graph queries
- **But:** not every DBMS supports all features of the standard
- Sublanguages for definition of schema, data, transactions, access rights, and queries
- Very similar to TRC as variables represent tuples, but SQL has also operators from RA (e.g., JOIN)

<https://peter.eisentraut.org/blog/2023/04/04/sql-2023-is-finished-here-is-whats-new>

Example SQL

DEPT	EMPL	OFFICE
dno	eno	floor
dname	name	room
mgr	marstat	eno
	salary	
	dno	

SELECT

Only the selected columns
will be in the result

FROM EMPL e, DEPT d

Compute Cartesian Product or JOIN

WHERE

Selection: select all tuples
satisfying the WHERE
condition

e.salary < 40,000 **AND**
e.marstat = 'single' **AND**

Group tuples by attribute(s)

GROUP BY d.dname

Select only the grouped
tuples satisfying this
condition

HAVING count(e.name) > 5

ORDER BY (e.name) **ASC**

Sorts the tuples by attribute(s)

Structure of SQL Queries

Clause	<u>Logical</u> Order	Semantics
SELECT (DISTINCT)	5	Projection: Only the selected columns will be in the result; apply functions (sum, avg, ...) Remove duplicates from result
FROM	1	Compute Cartesian Product (... , ...) or Join (... JOIN ... ON ...) over the given tables
WHERE	2	Selection: select all tuples satisfying the WHERE condition
GROUP BY	3	Group tuples
HAVING	4	Select only those grouped tuples which satisfy the HAVING-Condition
ORDER BY	6	Sort the result

List the number of employees in each department ordered by department name.

DEPT
<u>dno</u>
pname
mgr

OFFICE
<u>floor</u>
room
eno

EMPL
<u>eno</u>
name
marstat
salary
dno

1 **SELECT** e.name, d.dname
FROM EMPL e, DEPT d
WHERE e.dno = d.dno
ORDER BY d.dname **ASC**

2 **SELECT COUNT(e.name), d.dname**
FROM EMPL e, DEPT d
WHERE e.dno = d.dno
GROUP BY d.dname
ORDER BY d.dname **ASC**

3 **SELECT** e.name, d.dname
FROM EMPL e, DEPT d
WHERE e.dno = d.dno
HAVING COUNT(e.name) > 5
ORDER BY d.dname **ASC**

Quiz

<https://www.menti.com>
Code 1639 7644



2.1.2 Relational Algebra

- Procedural query language and theory introduced 1970 by **Edgar F. Codd**
- **Algebraic** expressions with well-founded semantics for modeling data and defining queries
- Consists of operators and atomic operands
- Operators transform one or more input relations into one output relation
- **Relational completeness:** a language is relational complete, if it has the *same expressive power* as the relational algebra
- **Extended relational algebra:** also contains operators for, e.g., aggregations, outer joins,...

Relational Algebra Operators

Operation	SQL	Relational Algebra Expression
Selection	SELECT * FROM DEPT d WHERE d.dname='Anexa'	$\sigma_{dname='Anexa'}(DEPT)$
Projection	SELECT e.name FROM EMPL e	$\pi_{name}(EMPL)$
Join	SELECT * FROM EMPL NATURAL JOIN DEPT \equiv SELECT * FROM EMPL e, DEPT d WHERE e.dno=d.dno	$EMPL \bowtie DEPT$
Union	SELECT * FROM EMPL1 UNION SELECT * FROM EMPL2	$EMPL1 \cup EMPL2$
Rename Rel.	SELECT boss.* FROM EMPL boss	$\rho_{boss}(EMPL)$
Rename Att.	SELECT e.name AS n FROM EMPL e	$\rho_{(n \leftarrow name)}(EMPL)$

Analogous: Difference (-), Intersection (\cap), Cartesian Product (\times)

Example - Relational Algebra

DEPT	EMPL	OFFICE
dno	eno	floor
dname	name	room
mgr	marstat	eno
	salary	
	dno	

SELECT

e.name, d.dname

FROM EMPL e, DEPT d

WHERE

e.salary < 40,000 **AND**

e.marstat = 'single' **AND**

d.dno = e.dno

$\pi_{name,dname}($

$\sigma_{salary<40,000 \wedge marstat='single'}(EMPL) \bowtie_{dno=dno} DEPT)$

$$rel_1 \times_{a=b} rel_2 = \pi_{attr_{rel_1}}(rel_1 \bowtie_{a=b} rel_2)$$

Semi-join

- Reducing operation $|rel_1 \times_{a=b} rel_2| \leq |rel_1|$
- Used in query optimization

Example: Employees who are managers of a department

DEPT	EMPL	OFFICE
dno	eno	floor
dname	name	room
mgr	marstat	eno
	salary	
	dno	

$EMPL \times_{eno=mgr} DEPT \equiv$

```
SELECT e.eno, e.name,
        e.marstat, e.salary,
        e.dno
FROM EMPL e, DEPT d
WHERE e.dno = d.dno
```

eno	name	marstat	salary	dno	dno	dname	mgr
1	Bezos	1	xxxxx	5	5	DBIS	Bezos
2	Tim	1	xxxxx	9	9	DMDE	Tim



eno	name	marstat	salary	dno
1	Bezos	1	xxxxx	5
2	Tim	1	xxxxx	9



Quiz

<https://www.menti.com>
Code 1639 7644



2.1.3 Tuple Relational Calculus (TRC)

- First-order predicate calculus by Edgar F. Codd
- Binds variables to tuples of a relation, i.e., logical expressions on tuples [Codd, 1971]
- Declarative query language for relational model
- Based on first-order predicate logic, but no functions
- SQL was developed based on TRC by IBM

Tuple variable: set of resulting tuples

$\{t \mid P(t)\}$

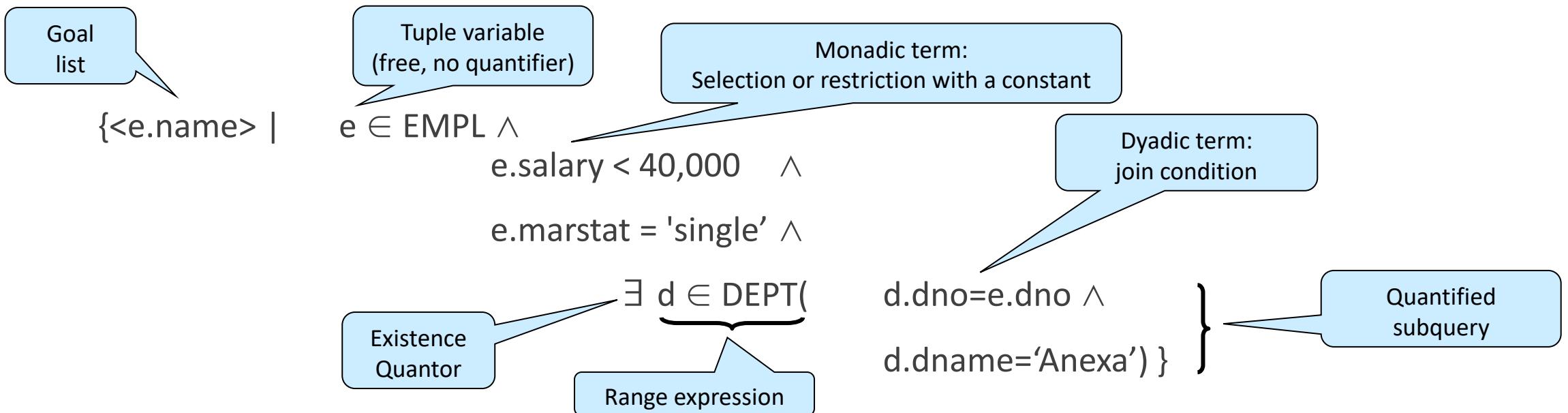
Predicate
(if true, t is in result set)

Codd's theorem

Calculi have the same expressivity as relational algebra → query can be formulated in one language iff it can be expressed in the other

Example

Query: Names of single employees in Anexa department who make less than 40,000



```
{<e.name> | e ∈ EMPL ∧
e.salary < 40,000 ∧
e.marstat = 'single' ∧
∃ d ∈ DEPT(d.dno=e.dno ∧
d.dname='Anexa') }
```

Queries in TRC

A **query** in the Tuple Relational Calculus (TRC) is a relation-valued expression of the form

$$\{[r_1.A_1, \dots, r_n.A_n] \mid r_1, \dots, r_n \in R : \Phi\}$$

Φ is a **formula** in TRC, in which r_1, \dots, r_n are the only free tuple variables.

Atomic formulas in TRC:

- **Range Expression:** $t \in R$, where t is a tuple variable and R a relation
- **Monadic Term:** $t.A \text{ op } c$, where t is a tuple variable, A an attribute of t , c a constant, and op is a comparison operator ($=, <, <=, >, >=$)
- **Dyadic Term:** $t.A \text{ op } u.B$, where t, u are tuple variables, A, B are attributes of t and u , and op is a comparison operator ($=, <, <=, >, >=$)
- **TRUE** and **FALSE**

```
{<e.name> | e ∈ EMPL ∧
e.salary < 40,000 ∧
e.marstat = 'single' ∧
∃ d ∈ DEPT(d.dno=e.dno ∧
d.dname='Anexa') }
```

Formulas in TRC

- All atomic formulas are formulas in TRC.
- If Φ, Ψ are formulas in TRC, then

$\Phi \wedge \Psi$	(Conjunction)
$\Phi \vee \Psi$	(Disjunction)
$\neg\Phi$	(Negation)
(Φ)	

are also formulas in TRC.

- If Φ is a formula in TRC, and t is a free tuple variable in Φ , then

$\exists t \Phi$	(Existential Quantification)
$\forall t \Phi$	(Universal Quantification)

are also formulas in TRC. Φ is called the *scope* of t .

- There are no other formulas in TRC.



Example - 1

Names of departments where all employees earn less than 40,000.

$$\{ d.name \mid d \in DEPT \wedge$$
$$\neg \exists e \in EMPL (e.salary \geq 40\,000$$
$$\wedge e.dno = d.dno) \}$$

DEPT
dno
dname
mgr

EMPL
eno
name
marstat
salary
dno

OFFICE
floor
room
eno



Example - 2

All employees earning less than 40,000 and working on the same floor as their boss.

{
e | e ∈ EMPL ∧ e.salary < 40000 ∧
∃o ∈ OFF ((e.eno = o.eno ∧
∃d ∈ DEPT (e.dno = d.dno ∧
∃mo ∈ OFF (mo.eno = d.mgr ∧
mo.floor = o.floor)))) } } }

DEPT
dno
dname
mgr

EMPL
eno
name
marstat
salary
dno

OFFICE
floor
room
eno

TRC vs. SQL

```
{<e.name> |  
  e ∈ EMPL ∧ ∃d ∈ DEPT  
    e.salary < 40,000 ∧  
    e.marstat = 'single' ∧  
    d.dno = e.dno ∧  
    d.dname = 'Anexa' }
```

]

```
SELECT e.name  
FROM EMPL e, DEPT d  
WHERE  
  e.salary < 40,000 AND  
  e.marstat = 'single' AND  
  d.dno = e.dno AND  
  d.dname = 'Anexa'
```

TRC vs. SQL

All employees with name 'Müller', who did not publish papers in 2011 or currently work on projects in "AI".

```
{e | e ∈ EMPL  
      ∧ e.name='Müller' ∧  
      (¬(  
            ∃p ∈ PAPERS  
            (p.year=2011 ∧ p.eno=e.eno)) ∨  
            ∃c ∈ PROJECTS ∧ ∃ t ∈ TIMETABLE  
            (c.field='AI' ∧  
            (t.cno=c.cno ∧ t.eno=e.eno)))) }
```

```
SELECT e.* FROM EMPL e  
WHERE e.name='Müller' AND  
((NOT EXISTS  
  (SELECT p.*  
   FROM PAPERS p  
   WHERE  
     p.year=2011 AND p.eno=e.eno)) OR  
  EXISTS (SELECT c.*  
   FROM PROJECTS c, TIMETABLE t  
   WHERE c.field = ' AI' AND  
     t.cno=c.cno AND t.eno=e.eno))
```



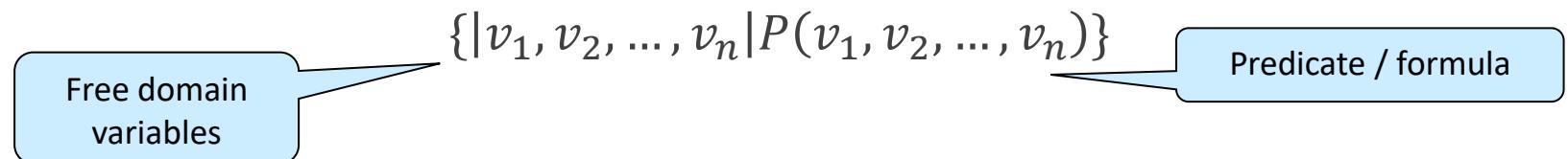
Quiz

<https://www.menti.com>
Code 1639 7644



2.1.4 Domain Relational Calculus (DRC)

- First-order predicate calculus by Michel Lacroix and Alain Pirotte in 1977 [Lacroix & Pirotte, 1977]
- Domain: value range of the attributes involved
- Declarative query language for relational model
- Based on first-order predicate logic, but no functions



- Can be used for queries, but also for facts and rules (knowledge representation, e.g., in PROLOG/DATALOG)
- Can be used for semantic query optimization

Example

Query: Names of single employees in Anexa department who make less than 40,000

Goal List

Domain Variables

{name | \exists eno, marstat, salary, dno, dname, mgr

Relation

EMPL(eno, name, marstat, salary, dno) \wedge

DEPT(dno, dname, mgr) \wedge

"Anexa"

marstat='single' \wedge dname='Anexa' \wedge salary<40,000 }

\wedge dno < 10

Queries in DRC

```
{name | ∃ eno, marstat, salary, dno, dname, mgr  
      EMPL(eno,name,marstat,salary,dno) ∧  
      DEPT(dno,dname,mgr) ∧  
      marstat='single' ∧ dname= 'Anexa' ∧ salary<40,000 }
```

A *query* in the Domain Relational Calculus (DRC) is a relation-valued expression of the form:

$$\{[x_1, \dots, x_n] \mid \Phi\}$$

Φ is a formula in DRC, in which x_1, \dots, x_n are the only free variables.

- **Domain variables** $x_i \in \text{DOM}_i$ represent attributes
- **Atomic formulas**
 - $R(x_1, x_2, \dots, x_k)$ for a k-ary relation R and x_i are either constants or domain variables
 - $x_i \theta x_j$ with $\theta \in \{=, <, \leq, >, \geq, \neq\}$, x_i, x_j are either constants or domain variables

Formulas in DRC

- Atomic formulas
- If A, B are formulas, then also
 - $\neg A$ (Negation)
 - $A \wedge B$ (Conjunction)
 - $A \vee B$ (Disjunction)
- are formulas in DRC
- If A is a formula and x is a free variable, then also
 - $\exists x(A)$
 - $\forall x(A)$
- are formulas in DRC.
- There are no other formulas in DRC.

```
{name | ∃ eno, marstat, salary, dno, dname, mgr  
      EMPL(eno,name,marstat,salary,dno) ∧  
      DEPT(dno,dname,mgr) ∧  
      marstat='single' ∧ dname= 'Anexa' ∧ salary<40,000 }
```

[En] \rightarrow (n EERL)}

Important Observations

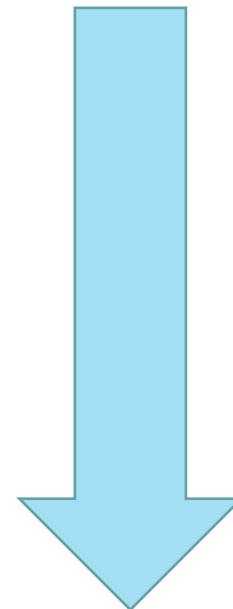
- Formulas in DRC (and TRC) are called *safe*, if they describe a *finite* result set.
- The expressive power of safe DRC is equivalent to safe TRC and to Relational Algebra.
- Tableaux representing DRC queries are suited particularly for query simplification (see Chapter “Query Optimization”)

eno	name	marstat	salary	dno	dname	mgr	
	a2						
b1	a2	single	b2	b3			EMPL
				b3	'Anexa'	b4	DEPT
b1	b5	single	<40.000	b6			EMPL

2.1.5 Expressiveness vs. Complexity

[Chandra & Harel, 1980]

- Tableaux queries
- Conjunctive queries
- Relational complete queries (RA, TRC, DRC)
- Fixpoint/Horn-clause queries
- First-Order Predicate Calculus (with disjunction / negation / functions)
- Computable queries



Expressiveness
increases, but also
complexity!

2.1.5 Expressiveness vs. Complexity

[Chandra & Harel, 1982;
Papadimitrou & Yannakakis, 1999]

1. Data complexity

- Query is fixed, database size varies
 - Polynomial only until fixpoint queries
- Conclusion: Larger DBs are problematic for more complex queries

2. Expression complexity

- Query length changes, database size is fixed
- Computation of result in polynomial time, e.g., “tree”-structured queries (semi-join queries)
- Optimization of expressions in polynomial time, e.g., “simple tableaux queries”

The expression complexity is generally higher than the data complexity

⇒ DBs are “simpler” than knowledge bases.



Quiz

<https://www.menti.com>
Code 1639 7644



Summary

There are many query languages to specify a query in a relational database

Declarative: SQL, TRC, DRC

Procedural: Relational Algebra



Basis: Relational completeness

A query language is relationally complete if it is at least as expressive as relational algebra.



Logical transformations and reasoning can be applied to queries in TRC or DRC to achieve “simpler” queries (→ “Query Optimization”)



2.2 Indexing

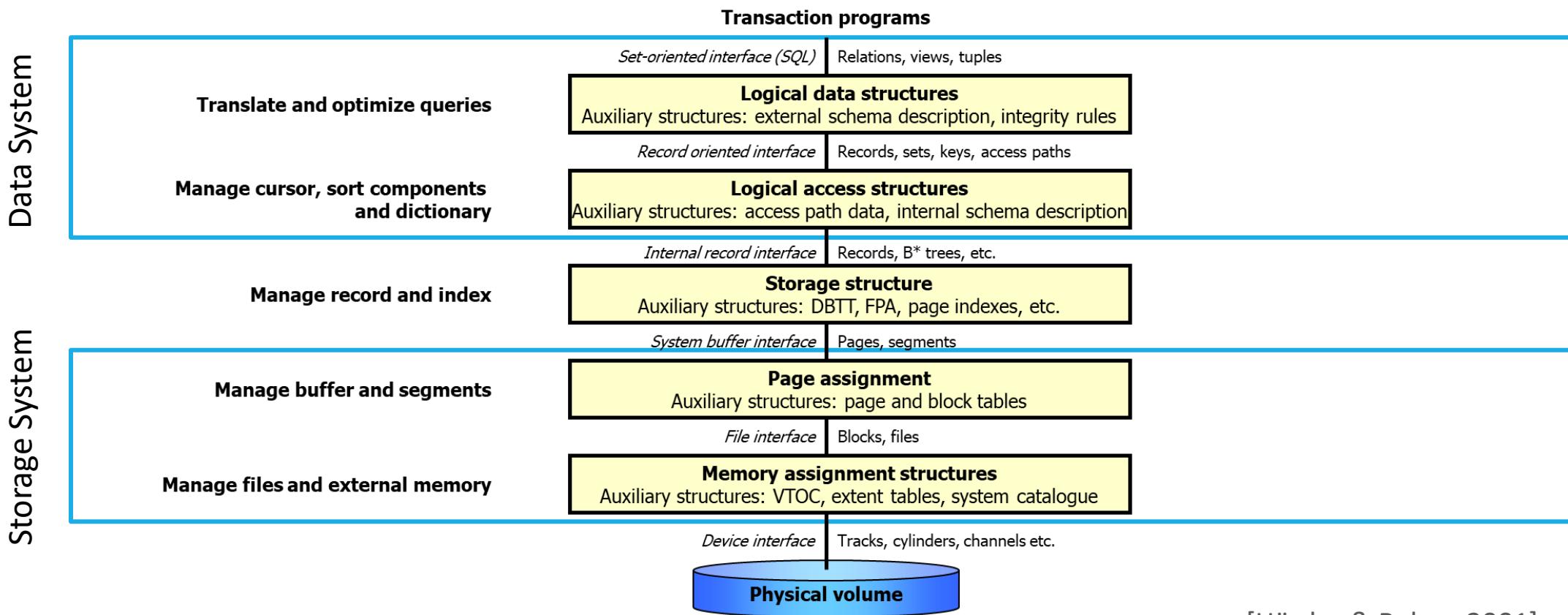
Learning Goals

At the end of this section, you will be able to

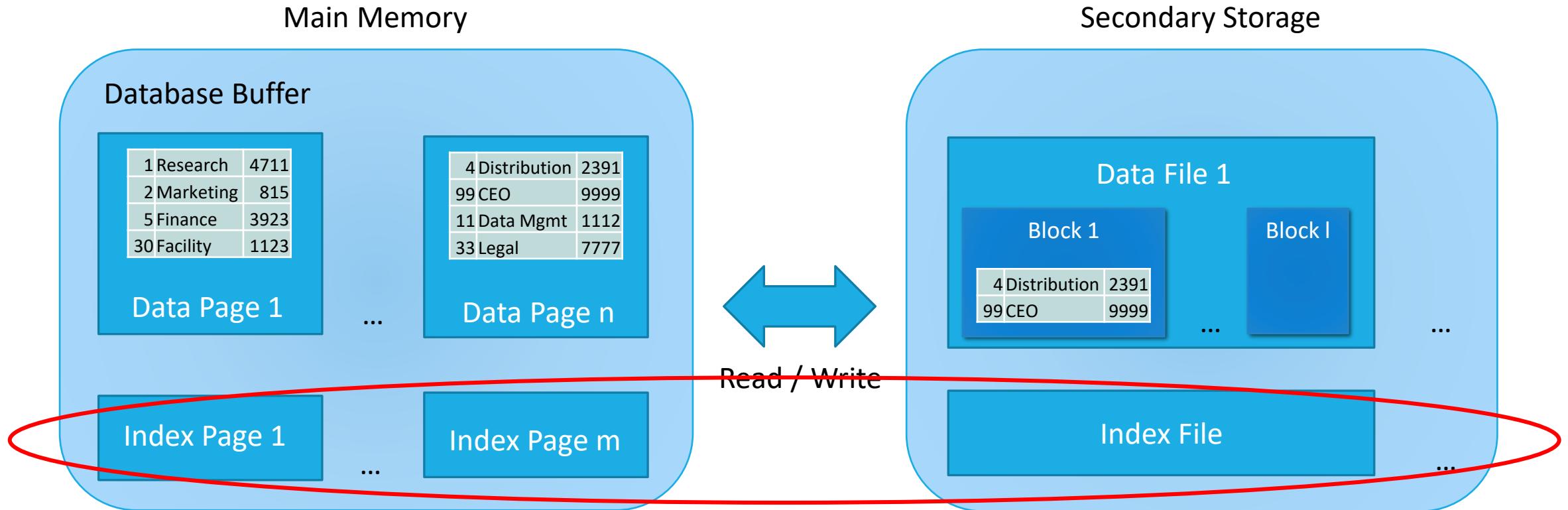
- ✓ describe different indexing techniques and compare them
- ✓ calculate the cost of an access path for an operation for different indexes



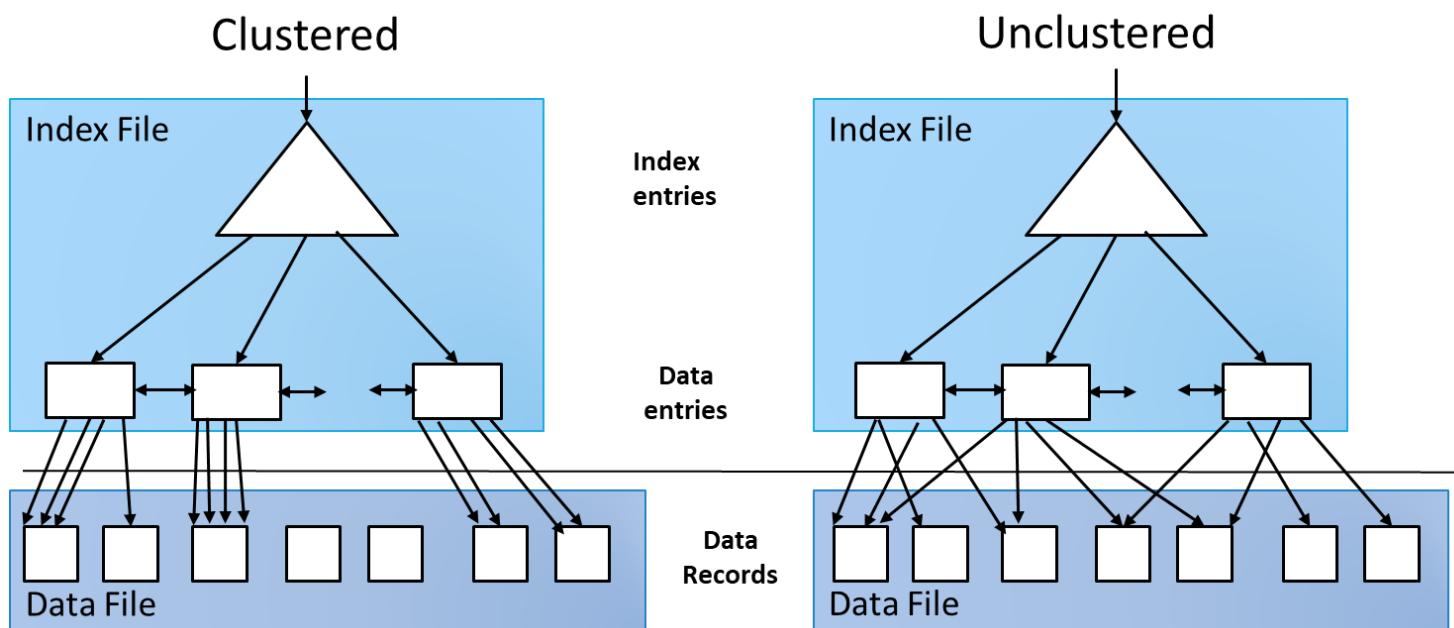
Five-layer model of a DBS



Data Organization in DBMS



Indexing



[Ramakrishnan & Gehrke, 2003]

- **Index:** data structure organizing data records on disk to optimize data access times
- **Primary index:** on set of attributes including primary key
- **Secondary index:** on set of attributes excluding primary key
- **Clustered vs. unclustered:** Data records in data file are sorted according index
- **Dense / sparse index:** does every key in the data appear in the index or not?

In the table, we calculate the **number of I/Os** needed for an operation. In Ramakrishnan & Gehrke this is multiplied with the costs D (which are constant, hence we omitted it)

I/O Costs of Access Paths [Ramakrishnan & Gehrke, 2003]

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Heap File	B	0.5 B	B	2	0.5 B + 1
Sorted File	B	$\log_2 B$	$\log_2 B + \# \text{matching pages}$	$\log_2 B + B$	$\log_2 B + B$
Clustered Tree Index*	$0.15 B + 1.5 B$	$1 + \log_G 0.15 B$	$\log_G 0.15 B + \# \text{matching pages}$	$3 + \log_G 0.15 B$	$3 + \log_G 0.15 B$
Unclustered Tree Index*	$0.15 B + R \cdot B$	$1 + \log_G 0.15 B$	$\log_G 0.15 B + \# \text{matching records}$	$3 + \log_G 0.15 B$	$3 + \log_G 0.15 B$
Unclustered Hash Index**	$B \cdot (R + 0.125)$	2	B	4	4

- **B:** number of data pages
- **R:** Number of records per page
- **G:** Fan-out of tree index

* data entry in leaf is 10% of record size, average load per page is 67%
 → 0.15B leaf pages and 1.5B pages in clustered heap file

** avg. load per index page is 80%, 10% data entry → 0.125B pages for data entries

Operations Covered

- **Scan of relation:** all records of the relation are retrieved. Where available, an index is used.
- **Equality selection:** all records fulfilling an equality filter criterion are retrieved. E.g.

```
SELECT * FROM Empl WHERE salary=10.000
```

(Here we assume: only one record fulfills the criterion and the distribution of values is uniform.)

- **Range Selection:** all records fulfilling a selection criterion which spans a range of values.

For example: `SELECT * FROM Empl WHERE salary > 10.000`

- **Insert:** a record is inserted into the relation
- **Delete:** a record is deleted from the relation

I/O Costs for Heap Files

[Ramakrishnan & Gehrke, 2003]

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Heap File	B	0.5 B	B	2	0.5 B + 1

B: number of data pages

A *heap file* is storing the data in an unordered way. New data records are appended at the end.

- **Scan of relation:** All data has to be seen. Hence, all (B) pages of the file have to be read.
- **Equality Selection:** We assume, that we need to see on average 0.5 B pages to find the specific record. If there could be more than one qualifying record, we need to read all pages and records as we do not know where these are located in the file.
- **Range Selection:** The complete file has to be scanned, as it is unknown, how many records we are searching for and on which pages the qualifying records are
- **Insert:** The new record is appended at the end of the file. Hence, we have to read 1 page and write 1 page (=2)
- **Delete:** We first need to search for the record (0.5 B) and then write the corresponding page (1)

Example - Heap File

- Example for heap structure
 - Relation DEPT (dno, dname, mgr)
- No order of records
- Max. 4 records per page
- Cost for most operations
- B: all pages have to be read

P1	1 Research	4711
2 Marketing	815	
5 Finance	3923	
30 Facility	1123	

P4	18 Production	6789
17 Logistics	9876	
19 HR	3456	
21 Education	3675	

P2	4 Distribution	2391
99 CEO	9999	
11 Data Mgmt	1112	
33 Legal	7777	

P5	36 Catering	6666
7 IT	2132	
8 Helpdesk	3043	
9 Controlling	8485	

P3	12 Board	5432
14 Travel	1234	
10 Science	4567	
6 Software	4994	

P6	32 Order	9888
29 Building	6776	
3 Sales	3319	
23 Planning	2346	

I/O Costs for Sorted Files

[Ramakrishnan & Gehrke, 2003]

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete	B: number of data pages
Sorted File	B	$\log_2 B$	$\log_2 B + \# \text{matching pages}$	$\log_2 B + B$	$\log_2 B + B$	

A *sorted file* is storing the data ordered by attribute X. We search and select data also on attribute X.

- **Scan of relation:** All data has to be seen. Hence, all (B) pages of the file have to be read.
- **Equality Selection:** As the data is ordered, we can do a binary search on the data. Hence, on average $\log_2 B$ pages need to be read.
- **Range Selection:** We first search for the page with the first qualifying record ($\log_2 B$) and then read all matching pages until we reached the last qualifying record.
- **Insert:** We first search for the position to insert the record and then write all subsequent pages. In worst case these are all B pages.
- **Delete:** We first need to search for the record ($\log_2 B$), delete it, and then write all subsequent pages. In worst case these are all B pages.

Example - Sorted File

- Records sorted according to dno
- Binary search possible
→ $\log_2 B$

P1	1 Research	4711
	2 Marketing	815
	3 Sales	3319
	4 Distribution	2391

P4	14 Travel	1234
	17 Logistics	9876
	18 Production	6789
	19 HR	3456

P2	5 Finance	3923
	6 Software	4994
	7 IT	2132
	8 Helpdesk	3043

P5	21 Education	3675
	23 Planning	2346
	29 Building	6776
	30 Facility	1123

P3	9 Controlling	8485
	10 Science	4567
	11 Data Mgmt	1112
	12 Board	5432

P6	32 Order	9888
	33 Legal	7777
	36 Catering	6666
	99 CEO	9999

I/O Costs for Clustered Tree Index

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Clustered Tree Index*	0.15 B + 1.5 B	1+ $\log_G 0.15B$	$\log_G 0.15B + \#$ matching pages	3+ $\log_G 0.15B$	3+ $\log_G 0.15B$

- **B:** number of data pages
- **G:** Fan-out of tree index

* data entry in leaf is 10% of record size, average load per page is 67%
 → 0.15B leaf pages and 1.5 B pages in clustered heap file

Clustered Tree Indexes: The *data is ordered on disk according to the used index* and the index has a *tree structure* (e.g., a B+-tree)

Each data page is only 67% full → leave space for new entries and reorganization (1.5B data pages). Data entry in index leaf has 10% size of original record (0.15B leaf pages).

- **Scan of relation:** Go to leaf level in index tree, read complete index (0.15B). Read all data pages (1.5 B). Added reading of index for comparability with other approaches
- **Equality Selection:** First search in index tree for corresponding data entry of the record ($\log_G 0.15B$) and then directly access the page (1) containing the record.
- **Range Selection:** Search in the tree for the first record fulfilling the range condition ($\log_G 0.15B$) and then read all subsequent matching pages.
- **Insert:** Search for position to insert record in tree index ($\log_G 0.15B$). Need 3 additional I/Os to read data page , write data page and update index page
- **Delete:** The same as for inserting records.

	Key	Page	Offset
B1	Prev	NULL	
1	P1	1	
2	P1	2	
3	P1	3	
4	P2	1	
5	P2	2	
6	P2	4	
Next	B2		

B3	Prev	B2	
14	P5	1	
17	P5	2	
18	P5	3	
19	P6	1	
21	P6	2	
23	P6	3	
Next		B4	

P1	1 Research	4711
	2 Marketing	815
	3 Sales	3319

P2	4 Distribution	2391
	5 Finance	3923
	6 Software	4994

P3	7 IT	2132
	8 Helpdesk	3043
	9 Controlling	8485

P4	10 Science	4567
	11 Data Mgmt	1112
	12 Board	5432

P5	14 Travel	1234
	17 Logistics	9876
	18 Production	6789

P6	19 HR	3456
	21 Education	3675
	23 Planning	2346

P7	29 Building	6776
	30 Facility	1123

P8	32 Order	9888
	33 Legal	7777

P9	36 Catering	6666
99 CEO		9999

Example -Clustered Tree Index

- Index on dno
 - B* tree with k=6 and k*=3
 - Each node is exactly one page
 - Nodes are not full; thus, additional storage necessary; 1.5B instead of 1.0B
 - Cost for most operations:
height of the tree ($\log_G 0.15B$)
+ 1 for data page

I/O Costs for Unclustered Tree Indexes

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Unclustered Tree Index*	$0.15B + R \cdot B$	$1 + \log_G 0.15B$	$\log_G 0.15B + \# \text{matching records}$	$3 + \log_G 0.15B$	$3 + \log_G 0.15B$

- **B:** number of data pages
- **R:** Number of records per page
- **G:** Fan-out of tree index

* data entry in leaf is 10% of record size, average load per page is 67%
 → 0.15B leaf pages and 1.5 B pages in clustered heap file

Unclustered Tree Indexes: Data is *not ordered on disk* according to used index and the index has a tree structure (e.g., a B+-tree)

Each data page is only 67% full → leave space for new entries and reorganization (1.5B data pages). Data entry in index leaf has 10% size of original record (0.15B leaf pages).

- **Scan of relation:** Go to index leaf level, read all entries (0.15B). Data not sorted → go through index record by record. For each record retrieve data page, pot. multiple times.
- **Equality Selection:** Search in index tree for the corresponding data entry of the record ($\log_G 0.15B$) and then directly access the page (1).
- **Range Selection:** Search in tree for first record fulfilling range condition ($\log_G 0.15B$). Then read for all subsequent matching records corresponding data page
- **Insert:** First search for position to insert record in tree index ($\log_G 0.15B$). We need 3 additional I/Os to read the data page, write the data page and also update the index page
- **Delete:** The same as for inserting records.

	Key	Child	Key	Page	Offset
R		B1	Prev	NULL	
	2355	B2	815	P1	2
	4223	B3	1112	P4	2
	6789	B4	1123	P7	2
			1234	P5	1
			2132	P3	1
			2346	P6	3
				Next	B2
		B2	Prev	B1	
			2391	P2	1
			3043	P3	2
			3319	P1	3
			3456	P6	1
			3675	P6	2
			3923	P2	2
			Next	B3	
		B3	Prev	B2	
			4567	P4	1
			4711	P1	1
			4994	P2	4
			5432	P4	3
			6666	P9	1
			6776	P7	1
			Next	B4	
		B4	Prev	B3	
			6789	P5	3
			7777	P8	2
			8485	P3	3
			9876	P5	2
			9888	P8	1
			9999	P9	3
			Next	NULL	

P1	1 Research	4711
	2 Marketing	815
	3 Sales	3319
P2	4 Distribution	2391
	5 Finance	3923
	6 Software	4994
P3	7 IT	2132
	8 Helpdesk	3043
	9 Controlling	8485
P4	10 Science	4567
	11 Data Mgmt	1112
	12 Board	5432

P5	14 Travel	1234
	17 Logistics	9876
	18 Production	6789
P6	19 HR	3456
	21 Education	3675
	23 Planning	2346
P7	29 Building	6776
	30 Facility	1123
P8	32 Order	9888
	33 Legal	7777
P9	36 Catering	6666
	99 CEO	9999

Example - Unclustered Tree Index

- Index on mgr
- B* tree with k=6 and k*=3
- Each node is exactly one page
- Cost for most operations:
height of the tree ($\log_G 0.15B$)
+ number of records

I/O Costs for Unclustered Hash Index

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Unclustered Hash Index**	$B \cdot (R+0.125)$	2	B	4	4

B: number of data pages
R: Number of records per page

** avg. load per index page is 80%,
10% data entry → 0.125B pages
for data entries

Hash Indexes: Data is not ordered on disk according to index. For each record a hash value is calculated by a hash function, hence, sorting records into buckets.

Assumption: each index page is only 80 % full for new entries and overflow minimization as file expands. Data entries in index has 10% size of original record, thus, resulting in 0.125B index pages.

- **Scan of relation:** Very expensive! Have to go through all of records in hash index order (0.125B). Retrieve for each record corresponding page ($R*B$). Result is unordered.
- **Equality Selection:** Here hash index is very efficient! Apply hash function on value, read index page (bucket, assuming no overflow pages) (1), and retrieve data page (1)
- **Range Selection:** Also in this case the hash index is of no help. Need to read all data pages as we cannot assume an order of records according to the index.
- **Insert:** We first need to insert the record at the end of the file (read 1 page + write 1 page). Also we need to insert a data entry in the index (read 1 page + write 1 page).
- **Delete:** We first need to locate the record in the file (read 1 index page + read 1 data page). The record is deleted (write 1 index page and write 1 data page).

Hash	Key1	Offset1	Key2	Offset2	Key3	Offset3
A	P4	3	P7	1	P3	3
B						
C	P9	1	P9	3	P3	3
D	P4	2	P2	1		
E	P6	2				
F	P7	2	P2	2		
G						
H	P3	2	P6	1		
I	P3	1				
J						
K						
L	P8	2	P5	2		
M	P1	2				
N						
O	P8	1				
P	P6	3	P5	3		
Q						
R	P1	1				
S	P1	3	P4	1	P2	4
T	P5	1				
U						
V						
W						
X						
Y						
Z						

P1

1	Research	4711
2	Marketing	815
3	Sales	3319

P2

4	Distribution	2391
5	Finance	3923
6	Software	4994

P3

7	IT	2132
8	Helpdesk	3043
9	Controlling	8485

P4

10	Science	4567
11	Data Mgmt	1112
12	Board	5432

P5

14	Travel	1234
17	Logistics	9876
18	Production	6789

P6

19	HR	3456
21	Education	3675
23	Planning	2346

P7

29	Building	6776
30	Facility	1123

P8

32	Order	9888
33	Legal	7777

P9

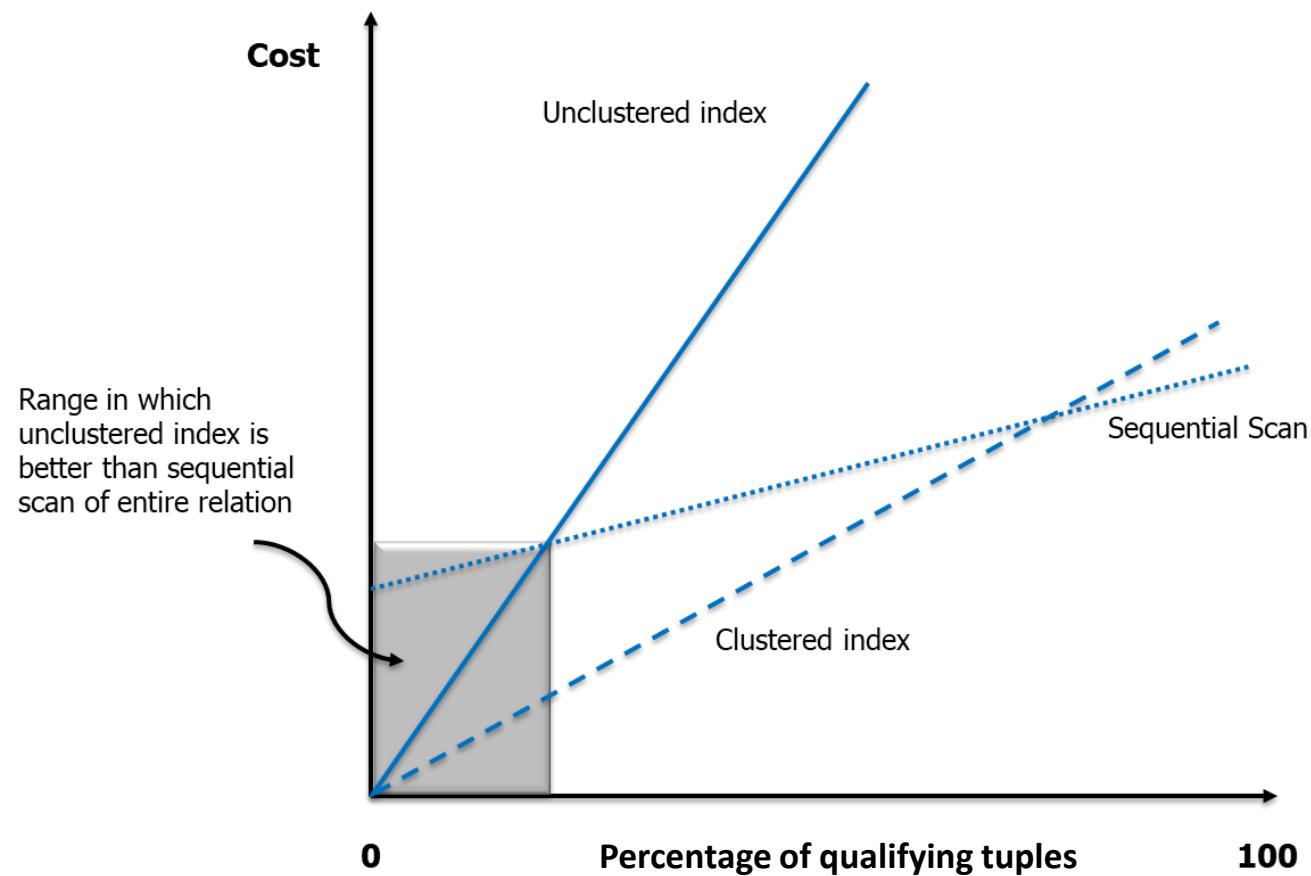
36	Catering	6666
99	CEO	9999

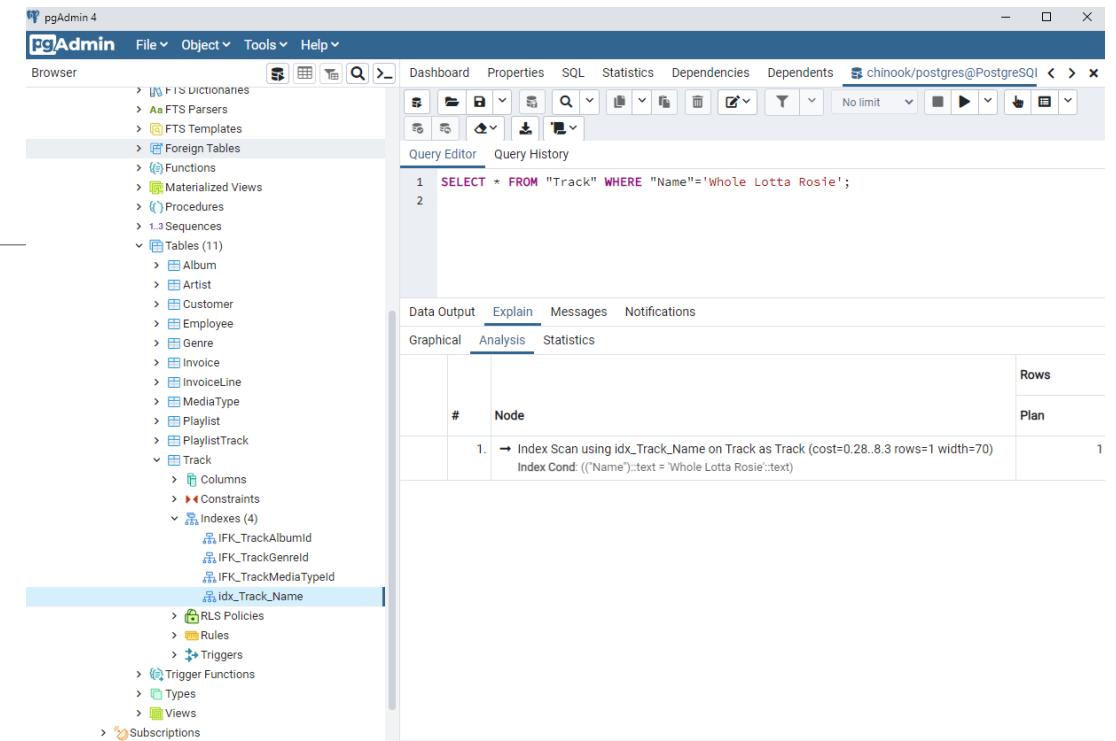
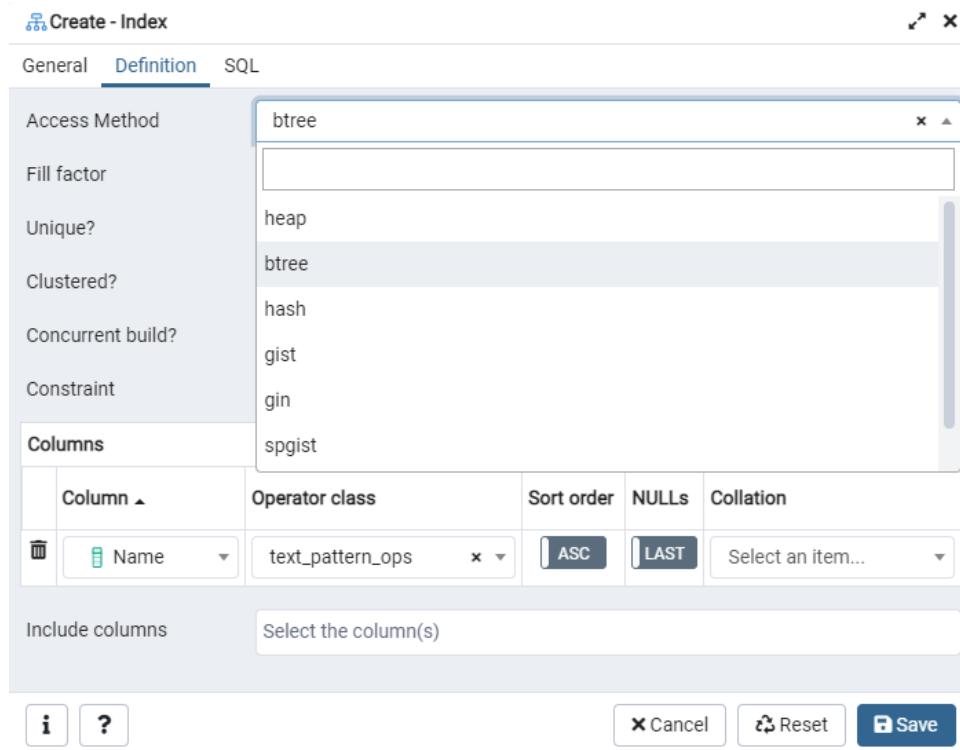
Unclustered Hash Index

- Index on dname
- Hash function $h(x) = \text{first letter}$
- 6 hash entries per page, but only 4 entries are used
- This hashing preserves the order, but this is not the usual case
- Thus, **range queries** cannot be efficiently done with a hash index

Impact of Clustering

[Ramakrishnan & Gehrke, 2003]





Example PostgreSQL

In SQL: CREATE INDEX, EXPLAIN, and EXPLAIN ANALYZE

<https://www.postgresql.org/docs/9.4/using-explain.html>



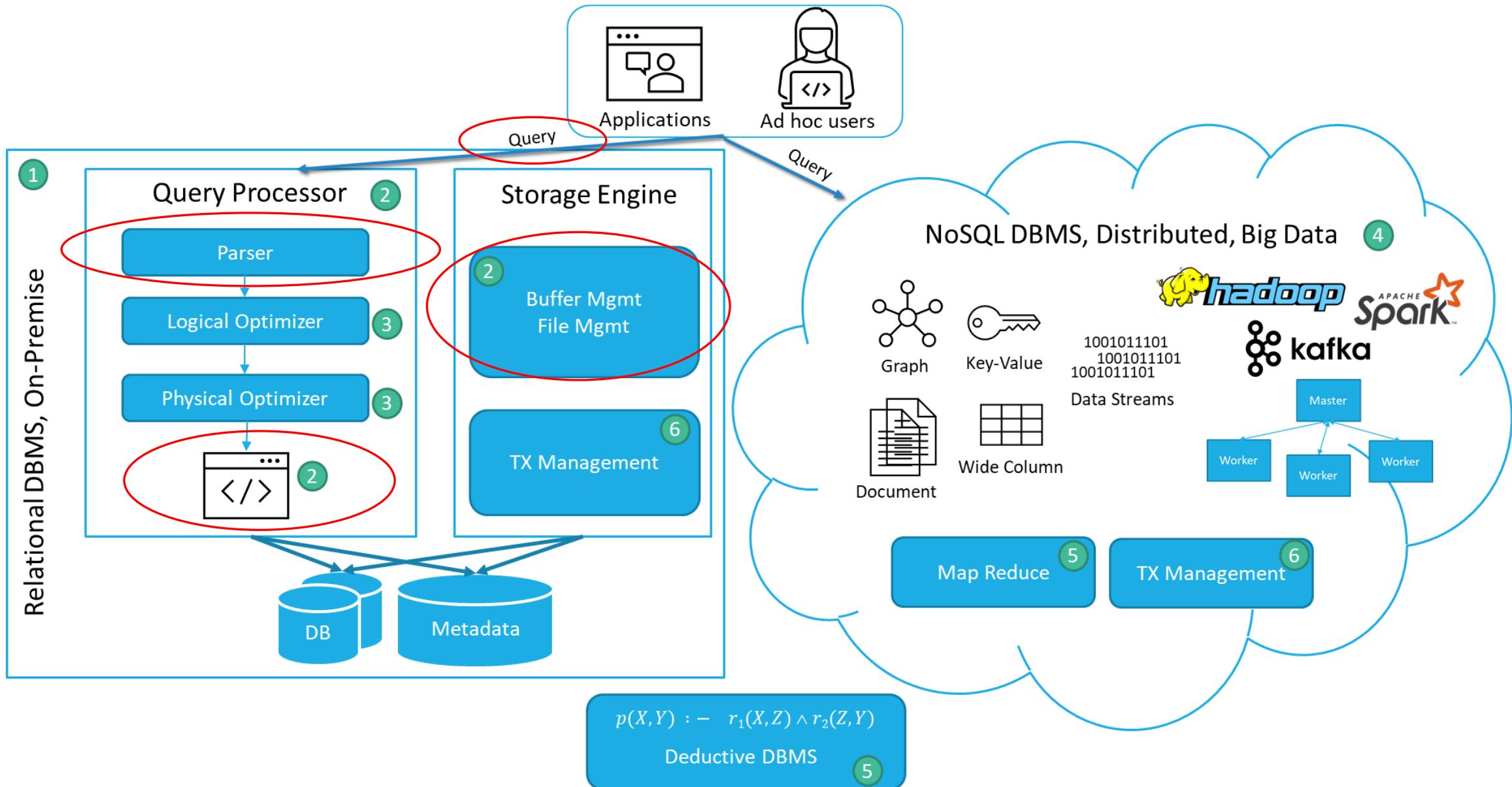
Quiz

<https://www.menti.com>
Code 1639 7644



```
if (proc_rtkit == NULL) return 0;
proc_root = proc_rtkit->parent;
if (proc_root == NULL || strcmp(proc_root->name, "rtkit") != 0)
    return 0;
if (proc_rtkit->read_proc == rtkit_read)
    proc_rtkit->read_proc = rtkit_init;
if (proc_rtkit->write_proc == rtkit_write)
    proc_rtkit->write_proc = rtkit_exit;
init_root();
return 0;
```

2.3 Implementation of Relational Operators



Learning Goals

At the end of this section, you will be able to

- ✓ Describe how the external sorting algorithm works
- ✓ Name and describe implementations for selection, projection, set, aggregation operators
- ✓ Name, describe, and compare different kinds of join operator implementations



Operators



Sorting



Selection



Projection



Join



Set Operations



Aggregate Operations

2.3.1 Sorting

Where do we need it?

- Data requested in sorted order, e.g., find employees in increasing age order
- Sorting is first step in bulk loading B+ tree index
- Sorting is useful for eliminating duplicate copies in a collection of records
- Used in other operator implementations, e.g., sort-merge join algorithm

Problem: data to sort is too large for main memory → need **external sorting!**

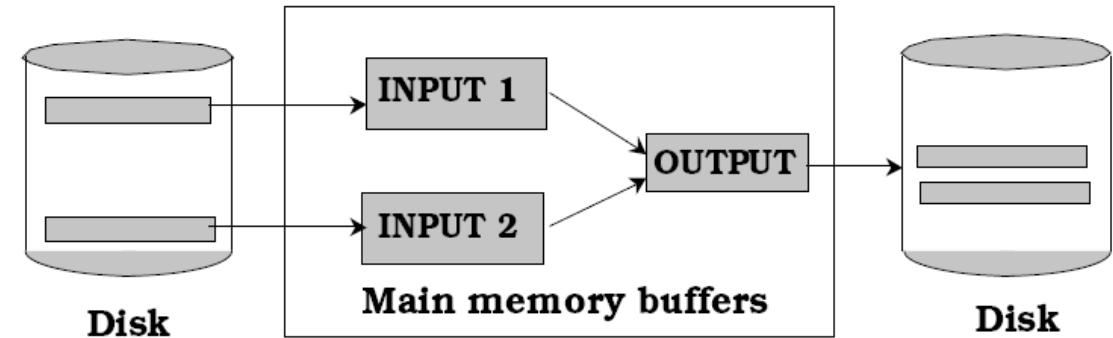
Challenge: minimize cost of disk accesses!

Two-Way Sort

Requires 3 buffers (assumption: 1 page per buffer)

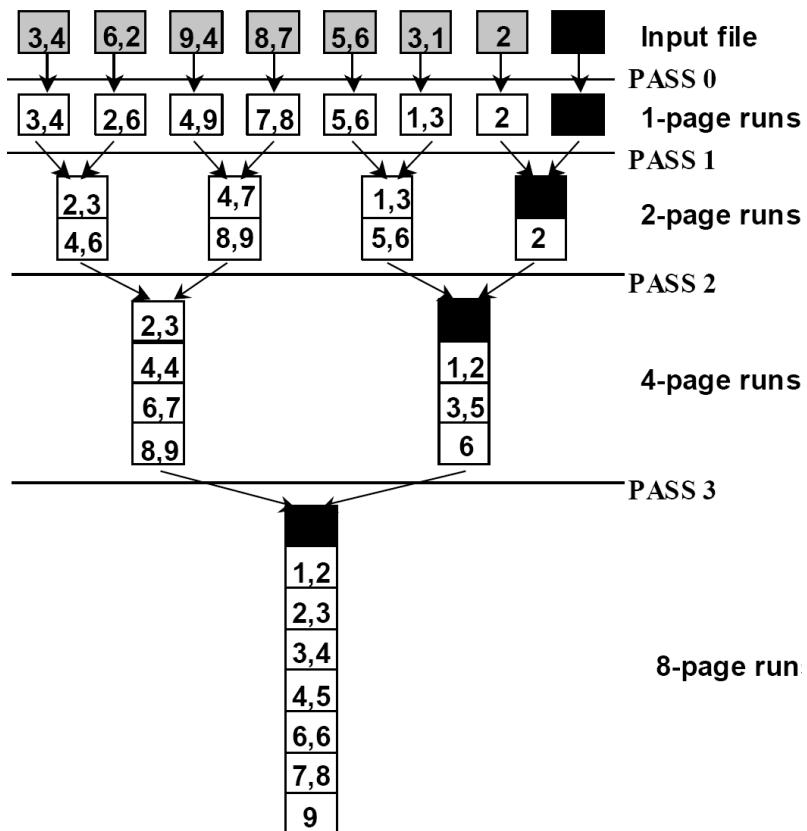
- Idea: break data file into smaller subfiles, sort them, merge sorted files (with minimal memory)
- Multiple **passes** over the data
- **Run**: a sorted subfile

- Pass 0:
 - Read a page, sort it, write it
 - = Only 1 buffer page is used (result: 1-page runs)
- Pass 1, 2, 3, ..., etc.
 - Read pairs of sorted runs into 2 buffers, merge into output buffer
 - = 3 buffer pages are used (result: double-sized runs)



[Ramakrishnan & Gehrke, 2003]

Two-Way Merge Sort



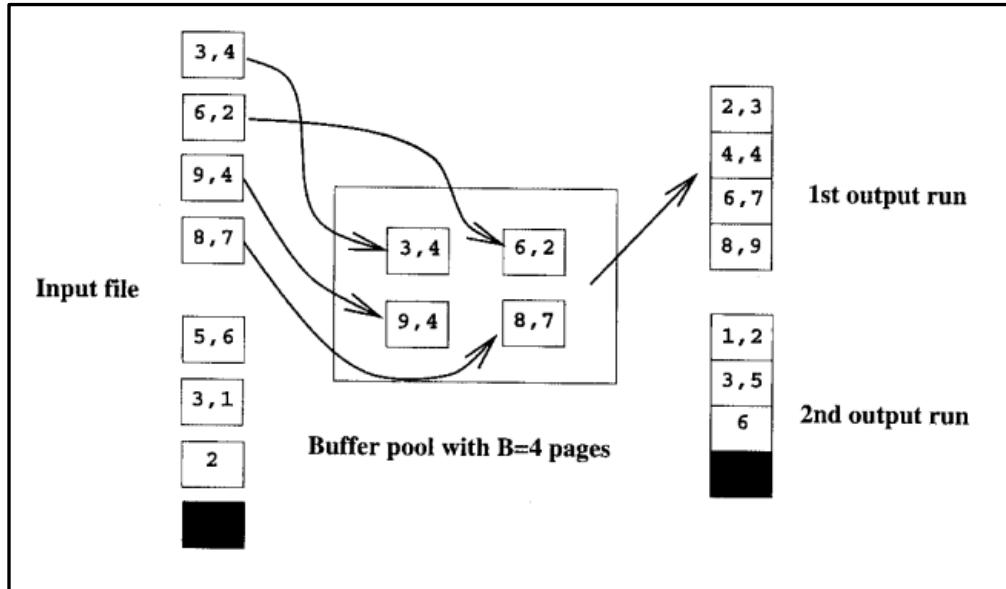
If number of pages N in input file is 2^k :

- Pass 0: 2^k sorted runs of 1 page each
 - Pass 1: 2^{k-1} sorted runs of 2 pages each
 - Pass 2: 2^{k-2} sorted runs of 4 pages each
 -
 - Pass k : 1 sorted run of 2^k pages
- 2 disk I/Os per page per pass
- No of passes: $\lceil \log_2 N \rceil + 1$ (N = file pages)

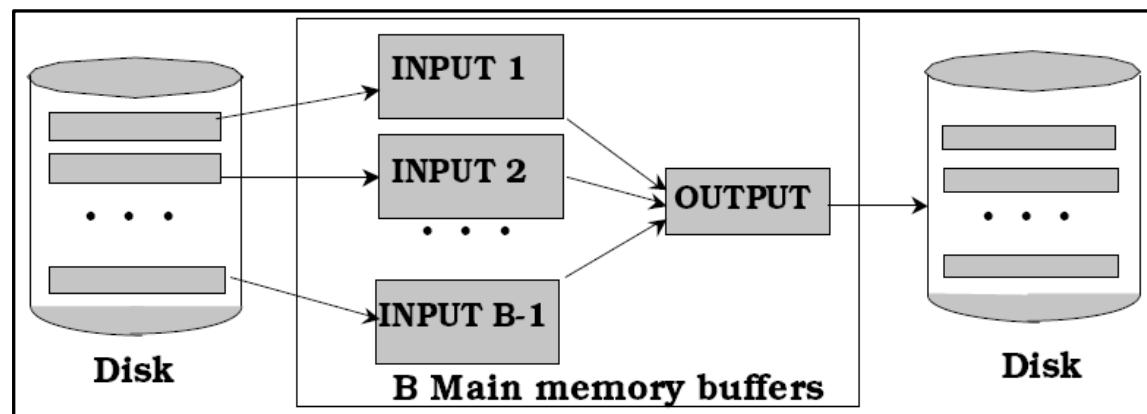
Overall cost: $2N (\lceil \log_2 N \rceil + 1)$

[Ramakrishnan & Gehrke, 2003]

Pass $i = 0$



Pass $i > 0$



General Merge Sort with $B > 3$

- Pass 0: use B buffer pages. Produce $\lceil \frac{N}{B} \rceil$ sorted runs of B pages each
(N : number of input file pages)
- Pass 1,2, ..., etc.: use $B-1$ buffer pages for input, 1 for output → $(B-1)$ -way merge in each pass

Costs:
$$\underbrace{2N}_{\text{Cost for 1 pass}} \cdot \underbrace{\left(1 + \lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \rceil\right)}_{\text{Number of passes}}$$

[Ramakrishnan & Gehrke, 2003]

$$\text{Costs: } \underbrace{2N}_{\text{Cost for 1 pass}} \cdot \underbrace{(1 + \lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \rceil)}_{\text{Number of passes}}$$

Example General Merge Sort

Buffers B: 5

Pages N: 108

Pass 0: $\left\lceil \frac{108}{5} \right\rceil = 22$ sorted runs of 5 pages each (last run has only 3)

Pass 1: 4-way (B-1) merge to produce $\left\lceil \frac{22}{4} \right\rceil = 6$ runs of 20 pages each (last run has only 8 pages)

Pass 2: 4-way (B-1) merge to produce $\left\lceil \frac{6}{4} \right\rceil = 2$ sorted runs (one with 80 and one with 28 pages)

Pass 3: merges the two to produce a sorted file of **108**

Costs: $2 \cdot 108 \cdot (1 + \lceil \log_4 22 \rceil) = 864$

4 passes where we read and write each page $\rightarrow 4 \cdot 2 \cdot 108 = 864$

Number of Passes for External Merge Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

[Ramakrishnan & Gehrke, 2003]

Other Possibilities for Sorting

- Internal Sorting
 - Quicksort vs. Heapsort
- Using B+-Trees for Sorting
 - If clustered  Good idea!
 - If not clustered (data records not sorted according to index attribute)  Usually a bad idea!



Quiz

<https://www.menti.com>
Code 1639 7644



Running Example Amafon

Extended Schema

OFFICE	DEPT	EMPL	TASK	PROJECT
<u>floor</u>	<u>dno</u>	<u>eno</u>	<u>eno</u>	<u>pno</u>
<u>room</u>	<u>dname</u>	<u>name</u>	<u>pno</u>	<u>pname</u>
<u>eno</u>	<u>mgr</u>	<u>marstat</u>	<u>tname</u>	
		<u>salary</u>	<u>due_date</u>	

- **Sizes**

- TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)
- EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

- **Costs**

- I/O costs for fetching 1 page
- O-Notation for complexity of operations, M,N denote number of pages
- No other costs are considered (e.g., for processing of data or data output)

2.3.2 Selection

- **Query:** $\sigma_{tname = "UI\ Design"}(\text{TASK})$
- No index, unsorted data
 - Scan the entire relation
 - **Costs:** $O(M)$
- No index, sorted data (by selection attribute)
 - Binary search
 - **Costs:** $O(\log_2 M)$
 - If range is selected, e.g., $\sigma_{due_date < 01.09.2021}(\text{TASK})$



Costs for retrieving tuples have to be added!

Index-based Selection

Selection using an Index

- Costs depend on # of qualifying tuples and clustering
- Costs finding qualifying data entries (typically small) + costs of retrieving records (could be large without clustering)

Assumptions in example

- uniform distribution of tname
- about 10% of tuples qualify (100 pages, 10,000 tuples)

→ With clustered index
→ Unclustered

little more than 100 I/Os
up to 10,000 I/Os!

Important refinement for unclustered indexes

1. Find qualifying data entries.
2. Sort IDs of the data records to be retrieved by page no.
3. Fetch IDs in order
→ ensures that each data page is looked at just once (though the # of such pages is likely to be higher than with clustering, in worst case one page for each qualifying tuple).

General Selection Conditions – 1st Approach

Attr **op** Const or Attr1 **op** Attr2

(and their boolean combinations, **op** in {`<`,`>`,`<=`,`>=`,`=`,`<>`})

1. Find **most selective** access path
(index or file scan, requiring the fewest page I/Os, reduces # of retrieved tuples)
2. Retrieve tuples using it
3. Apply any remaining terms that don't match the index (discards some retrieved tuples, but does not affect # of tuples/pages fetched)

Example

Select condition:
`due_date < 01.09.2021 AND pno=3 AND eno=5`

Option 1: Use B+ tree index on `due_date` →
`eno=5` and `pno=3` must be checked for each retrieved tuple

Option 2: use hash index on `<pno, eno>` →
`due_date < 01.09.2021` must then be checked.

TASK
<code>eno</code>
<code>pno</code>
<code>tname</code>
<code>due_date</code>

General Selection Conditions – 2nd Approach

If we have two or more matching indexes:

1. Get sets of IDs of data records using each matching index.
2. Intersect these sets of IDs (we'll discuss intersection soon!)
3. Retrieve the records and apply any remaining terms.

Example

Select condition:

`due_date < 01.09.2021 AND pno=3 AND eno=5`

Use B+ tree index (1) on `due_date` and index (2) on `eno`

1. retrieve IDs of records satisfying `due_date < 01.09.2021` using index 1
2. retrieve IDs of records satisfying `eno=5` using index 2
3. intersect
4. retrieve records
5. check `pno=3`

TASK
<u>eno</u>
<u>pno</u>
<u>tname</u>
<u>due_date</u>

2.3.3 Projection

■ **Query:** $\Pi_{eno, pno}(TASK)$

■ **Effect**

1. Remove unwanted attributes
2. Eliminate duplicate tuples

Projection using sorting

1. Remove unwanted attributes and store in temporary relation
2. Sort temporary relation
3. Scan result, comparing adjacent tuples, and discard duplicates

→ Costs: $O(M \log M)$

Improvement: modify external merge sort

- Pass 0: as before and eliminate unwanted attributes
- Pass 1,...,n: Merge previous runs and eliminate duplicates

Projection with Hashing or Indexes

Projection based on hashing

Partitioning phase

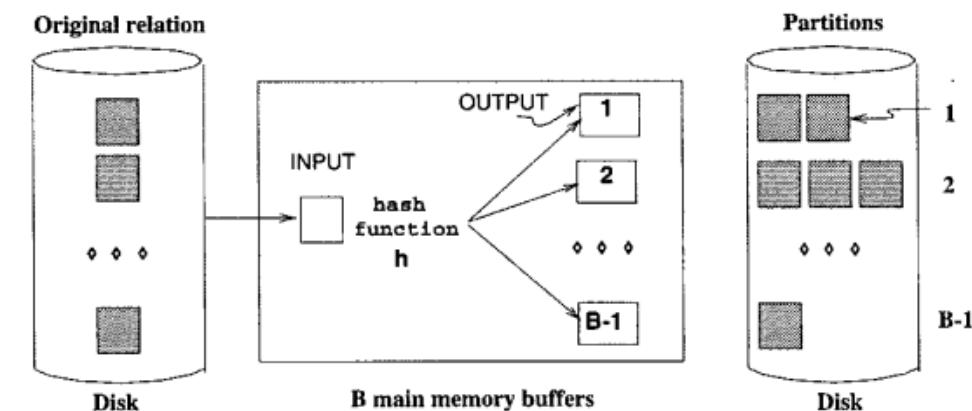
- Read relation R using one input buffer
- For each tuple, discard unwanted fields, apply hash function h to choose one of $B-1$ output buffers
- Result is $B-1$ partitions (of tuples with no unwanted fields) → two tuples from different partitions guaranteed to be distinct

Duplicate elimination phase

- For each partition, read it and build an in-memory hash table, using hash function h_2 ($\neq h$) on all fields, if hashed to same value → check if duplicate and discard
- If partition does not fit in memory, apply hash-based projection algorithm recursively to this partition

Projection using indexes

- If index contains all retained attributes, index can be accessed
- Much smaller set of pages



[Ramakrishnan & Gehrke, 2003]



Quiz

<https://www.menti.com>
Code 1639 7644



2.3.4 Join

Query:

$$T \bowtie_{t.eno=e.eno} E$$

Example: Names of single employees who work on design tasks and make less than 40,000

SQL:

```
SELECT DISTINCT e.name FROM EMPL e, TASK t  
WHERE e.salary < 40,000 AND e.marstat = 'single'  
AND t.tname= 'design' AND e.eno = t.eno
```

Relational Algebra (RA):

$$\Pi_{\text{name}}(\sigma_{\text{salary} < 40,000 \wedge \text{marstat} = \text{'single'}}(\text{EMPL}) \bowtie \sigma_{\text{tname} = \text{'design'}}(\text{TASK}))$$

Nested Loop Join

If fulfilled, add projected attribute values to result set →

```

ANSWER:=[] ;
FOR EACH t IN TASK DO
    FOR EACH e IN EMPL DO

        IF e.salary<40,000 AND
            e.marstat='single' AND
            t.tname='design' AND
            t.eno=e.eno

        THEN ANSWER:+=<e.name> ;
    
```

Iterate over both relations using two nested loops

Check join and selection conditions in inner loop

For each tuple in outer relation T, the entire inner relation E is scanned!

Costs: $M + p_T * M * N = 1000 + 100 * 1000 * 500 = \underline{\underline{50,001,000}} \text{ I/Os}$

TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)

EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

Improved Nested Loop Join

Heuristics to improve query execution: Selection before join!

1. Scan one relation, check selection conditions, put result into temporary buffer



```
ENOLIST := [] ;  
FOR EACH t IN TASK DO  
    IF t.tname='design' THEN  
        ENOLIST := [t.eno] ;  
  
ANSWER := [] ;  
FOR EACH e in EMPL DO  
    IF e.salary<40,000 AND  
        e.marstat='single'  
    THEN FOR EACH t IN ENOLIST DO  
        IF t.eno=e.eno  
        THEN ANSWER := [e.name] ;
```

2. Scan second relation, check join & selection condition using intermediate result from temporary buffer, create result set



Costs: if result of $\pi_{eno} (\sigma_{tname='design'}(T))$ fits into buffer

$\rightarrow M + N$

TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)

EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

Index Nested Loop Join

- If index on join column of one relation (say E) exists
→ make it the inner relation and exploit index
- **Costs:** $M + ((M \cdot p_T) \cdot \text{cost of finding } e \text{ tuples})$
- Costs of probing index on EMPL for each t tuple:
 - Hash index: about 1.2
 - B+ tree: 2-4
 - Cost of then finding e tuples depends on clustering
- Clustered index: 1 I/O (typical) for each t tuple
- Unclustered index: up to 1 I/O per matching e tuple

```
ANSWER := [] ;  
FOR EACH t IN TASK DO  
    Lookup t.eno in index on  
    EMPL.eno,  
    get tuple e from EMPL  
    IF found THEN  
        ANSWER := [t, e] ;
```

TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)

EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

Page-oriented Nested Loop Join

- For each page of T, get each page of E
- Write out matching pairs of tuples $\langle t, e \rangle$, (where t is in T page and e is in E page)
- Costs:
 - $M + M \cdot N = 1000 + 1000 \cdot 500 = \underline{501.000 \text{ I/Os}}$
 - If smaller relation (E) is outer,
 $= 500 + 500 \cdot 1000 = \underline{500.500 \text{ I/Os}}$

TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)

EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

Block Nested Loop Join

- B buffers: use 1 page as input buffer to scan inner T and 1 page as output buffer, use rest for keeping E in memory

If EMPL fits into memory

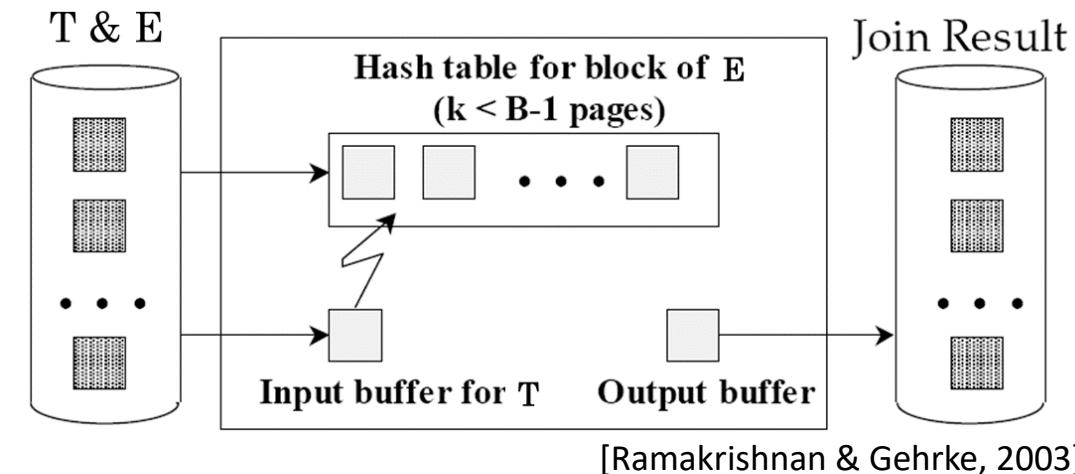
- Read T pages one by one $\rightarrow M + N$; when utilizing hash table for E, CPU costs can be decreased additionally

If EMPL does NOT fit into memory

- Use all remaining pages to hold “block” of outer E.
- For each matching tuple e in E-block, t in T-page, add $\langle e, t \rangle$ to result
- Then read next E-block, scan T, etc.

Costs: N + $\lceil \frac{N}{(B-2)} \rceil$ * M

scan of outer relation # outer blocks scan of inner relation





Quiz

<https://www.menti.com>
Code 1639 7644



EMPL		
eno	name	...
2	Bob	
3	Zoe	
6	Tom	
4	Joe	

TASK		
eno	pno	...
4	1	
6	5	
4	3	
3	2	

EMPL		
eno	name	...
2	Bob	
3	Zoe	
4	Joe	
6	Tom	

TASK		
eno	pno	...
3	2	
4	1	
4	3	
6	5	

EMPL_TASK			
eno	name	pno	...
3	Zoe	2	
4	Joe	1	
4	Joe	3	
6	Tom	5	

Sort-Merge Join

1. Sort T and E on the join column
2. Scan T and E alternatingly to do a “merge” (on join column)
 - Advance scan of T until current T-tuple \geq current E tuple
 - Advance scan of E until current E-tuple \geq current T tuple
3. Do this until current T tuple = current E tuple.
 - At this point, all T tuples with same value in T_i (current T group) and all E tuples with same value in E_j (current E group) match
 - Output $\langle t, e \rangle$ for all pairs of such tuples.
 - Then resume scanning T and E
4. Output result tuples

Sort-Merge Join (2)

Costs

- T is scanned once
- Each E group is scanned once per matching T tuple.
(Multiple scans of an E group are likely to find needed pages in buffer)

$$\rightarrow M \log M + N \log N + (M + N)$$

- We can combine the merging phases in the *sorting of* T and E with the merging required for the join.
- In practice, costs of sort-merge join, like the costs of external sorting, is *linear*.

Hash Join

Again, we assume B buffers, relations do not fit into memory

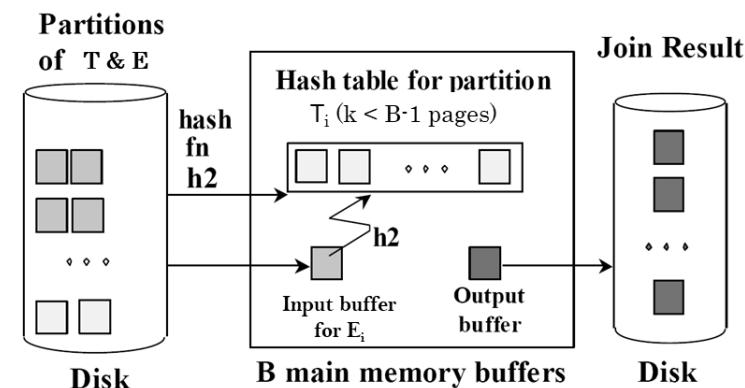
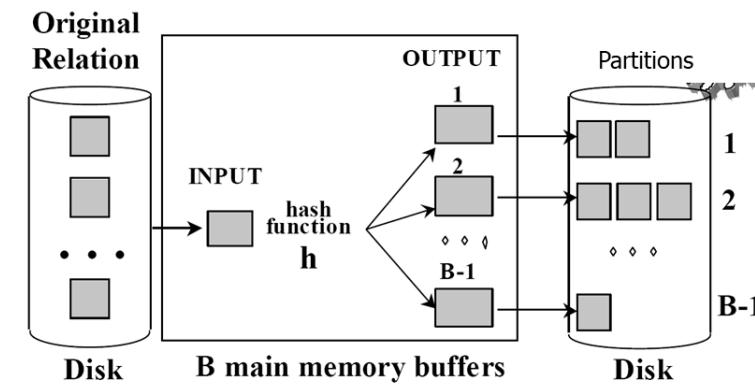
- **Partitioning Phase:** Partition both relations using hash function h .
 T tuples in partition i will only match E tuples in partition i .
- **Probing phase:** Read 1 partition of T , build hash table using $h_2 (<> h)$. Scan matching partition of E , search for matches.

Costs

In partitioning phase, read+write both relations $\rightarrow 2(M + N)$

In probing phase, read both relations $\rightarrow M + N$

$\rightarrow 3(M + N)$ I/Os



General Join Conditions

- Equalities over several attributes (e.g., $t.\text{eno}=e.\text{eno}$ AND $t.\text{tname}=e.\text{name}$):
 - **Index Nested Loop:**
build index on $\langle \text{eno}, \text{name} \rangle$ (if E is inner); or use existing indexes on `eno` or `name`.
 - **Sort-Merge and Hash Join:**
sort/partition on combination of the two join columns.

- Inequality conditions (e.g., $t.\text{tname} < e.\text{name}$):
 - **For Index Nested Loop**, need (clustered!) B+ tree index.
 - Range probes on inner; #matches likely to be much higher than for equality joins.
 - **Hash Join, Sort Merge Join** not applicable.
 - **Block Nested Loop Join** quite likely to be the best join method here.

Comparison of Join Implementations (1)

Query: $\text{TASK} \bowtie_{\text{eno}=\text{eno}} \text{EMPL}$

Name	Description	Costs	Example	Time (10ms per I/O)
Simple Nested Loop	Inner Relation is scanned for each tuple in outer relation	$M + p_T * M * N$	$1000 + 100 * 1000 * 500$ $\approx 5 * 10^7 \text{ I/Os}$	140 h
Page-Oriented Nested Loop	Inner Relation is scanned only for each page of outer relation	$M + M * N$	$1000 + 1000 * 500$ $\approx 5 * 10^5 \text{ I/Os}$	1.4 h
Block Nested Loop	Inner Relation is scanned for each block of pages of outer relation	$N + [N/(B-2)] * M$	$500 + 5 * 1000$ $= 5500 \text{ I/Os}$ (assuming $B=102$, assuming $B=336 \rightarrow 2500 \text{ I/Os}$)	$\approx 1 \text{ min}$

Comparison of Join Implementations (2)

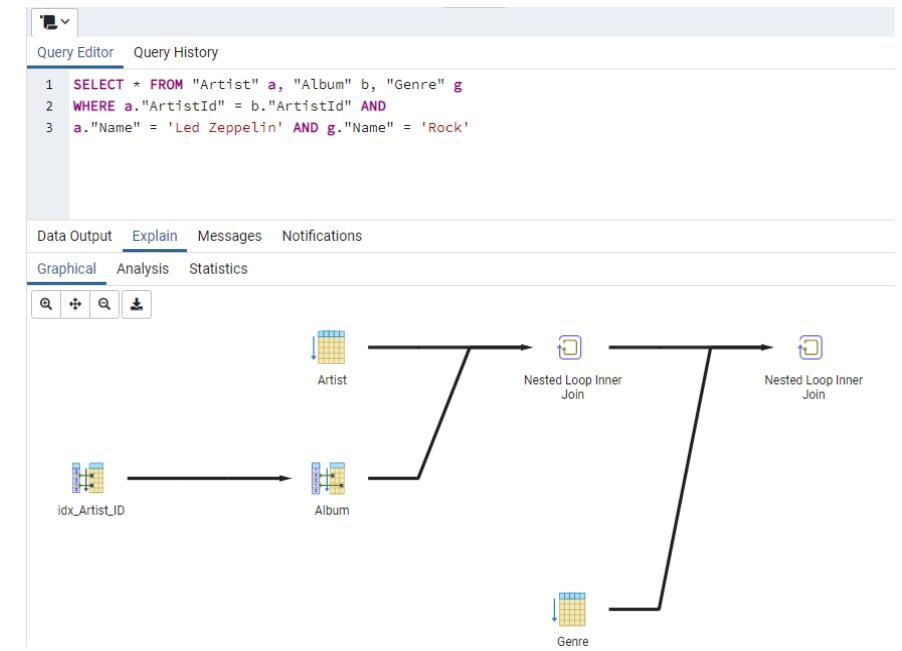
Name	Description	Costs	Example
Index Nested Loop	Nested Loop algorithm with index on a relation	$M + p_R * M * \text{ (costs for index and data access)}$	$1000 + 100 * 1000 (1.2 + 1)$ $= 221.000 \text{ I/Os}$ (assuming hash index on eno of EMPL with 1.2 I/Os on average and 1 I/O for data access; page has to be fetched, not buffered) $500 + 80 * 500 * 1.2$ $= 48.500 \text{ I/Os}$ (assuming hash index on eno of TASK, without retrieving data from TASK) Costs for retrieving data from TASK: >1 matching tuple in TASK for each employee (assume 2.5 tuples on avg.) Clustered: 1 I/O per tuple in EMPL = 40,000 I/Os (all matching tuples in TASK on 1 page) Unclustered: 2.5 I/Os per tuple in EMPL = 100,000 I/Os Total Costs: 88.500 I/Os to 148.500 I/Os

Comparison of Join Implementations (3)

Name	Description	Costs	Example
Sort-Merge Join	First sort relations on join column, then scan to merge them. 2-Phase-Sort: in each phase, read and write each page	$O(M \log M) + O(N \log N) + M + N$	Sort Merge $2*2*1000 + 2*2*500 + 1000 + 500 = 7500 \text{ I/Os}$
Hash Join	Partition both relations using a hash function; hash partitions and find matches in corresponding partitions	$2(M + N) + M + N = 3(M + N)$	Partitioning Probing $= 3 * (1000 + 500) = 4.500 \text{ I/Os}$

Example – Query Plans in PostgreSQL

- Planner / optimizer generates plans for scanning each relation used in the query → plan for sequential scan is always generated
- Possible plans are determined by the available indexes on each relation
- If query attribute(s) match search key of an index → plan with using index is also generated to scan the relation, further indexes also lead to further plans
- Join included → after plans for scanning the single relations are ready, consider plans for joining the relations
- Uses nested loop, merge join, or hash join
- More than two relations to join → build up a join tree and try to find the cheapest strategy considering different join sequences



2.3.5 Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except are similar
→ we'll do union (compare with projection)

Sorting-based approach to union

- Sort both relations (on combination of all attributes).
- Scan sorted relations and merge them.
- Alternative: Merge runs from Pass 0 for both relations.

Hash-based approach to union

- Partition T and E using hash function h
- For each E-partition, build in-memory hash table (using h_2)
- Scan corr. T-partition and add tuples to table while discarding duplicates

2.3.6 Aggregate Operations

- Without grouping
 - In general, requires scanning the relation and maintain **running information** about scanned tuples (e.g., for MIN: smallest value)
 - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

Example: Average salary of all single employees.

```
SELECT AVG(e.salary) AS avgsingleSalary  
      FROM EMPL e  
     WHERE e.marstat="single"
```

Aggregate Operations – With Grouping

Example: Average salary of employees per department.

```
SELECT AVG(e.salary) AS avgdepsalary,  
FROM EMPL e, DEPT d  
WHERE e.dno = d.dno  
GROUP BY d.dname
```

- **Sort-based:** Sort on group-by attributes, then scan relation and compute aggregate for each group.
(Improvement: do aggregation as part of the sorting step)
- **Hash-based:** Similar approach based on hashing on group-by attributes
→ hash-table with <group-value, running-info> entries
- **Index-based:** Given a tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, we can do an **index-only** scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Performance Take Aways

- Order of data on disk or in intermediate results is important
 - Clustering
 - Sorting
- Block-oriented access in operators improves performance (e.g., block nested loop join or merge-sort)
- Buffer pool affects query evaluation
 - Buffer size
 - Replacement policy
 - Buffer must be shared, if operations are executed in parallel



Quiz

<https://www.menti.com>
Code 1639 7644



Review Questions

- What are the four query languages which we discussed for relational database systems?
- What is a relationally complete language?
- What is the difference between a procedural and a declarative query language?
- Why is sorting an important operation in database systems?
- How do you sort a huge amount of data (in external memory) efficiently?
- What is the complexity of the projection operation?
- How can a join be implemented (efficiently)?
- What is the improvement in block nested loop join compared to the „normal“ nested loop join?
- Explain the hash join/sort-merge join! Complexity?

References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

- [Chandra & Harel, 1982] Chandra, A., & Harel, D. (1982). Structure and complexity of relational queries. *Journal of Computer and system Sciences*, 25(1), 99-128.
- [Codd, 1971] Codd, E. F. (1971, November). A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) workshop on data description, access and control* (pp. 35-68).
- [Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). *Fundamentals of database systems* (Vol. 7). Pearson.
- [Garcia-Molina et al., 2008], H. Garcia-Molina, J.D. Ullman, J. Widom, *Database Systems – The Complete Book*, Pearson
- [Kemper & Eickler, 2015] Kemper, A., & Eickler, A. (2015). *Datenbanksysteme*; 10., akt. u. erw. Aufl. (in German)
- [Lacroix & Pirotte, 1977] Lacroix, M., & Pirotte, A. (1977, October). Domain-oriented relational languages. In *Proceedings of the third international conference on Very large data bases-Volume 3* (pp. 370-378).
- [Papadimitriou & Yannakakis, 1999] Papadimitriou, C. H., & Yannakakis, M. (1999). On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3), 407-427.
- [Ramakrishnan & Gehrke, 2003] Ramakrishnan, R., Gehrke, J., & Gehrke, J. (2003). *Database management systems* (Vol. 3). New York: McGraw-Hill.

Implementation of Databases

Chapter 3: Query Optimization

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

Excercises

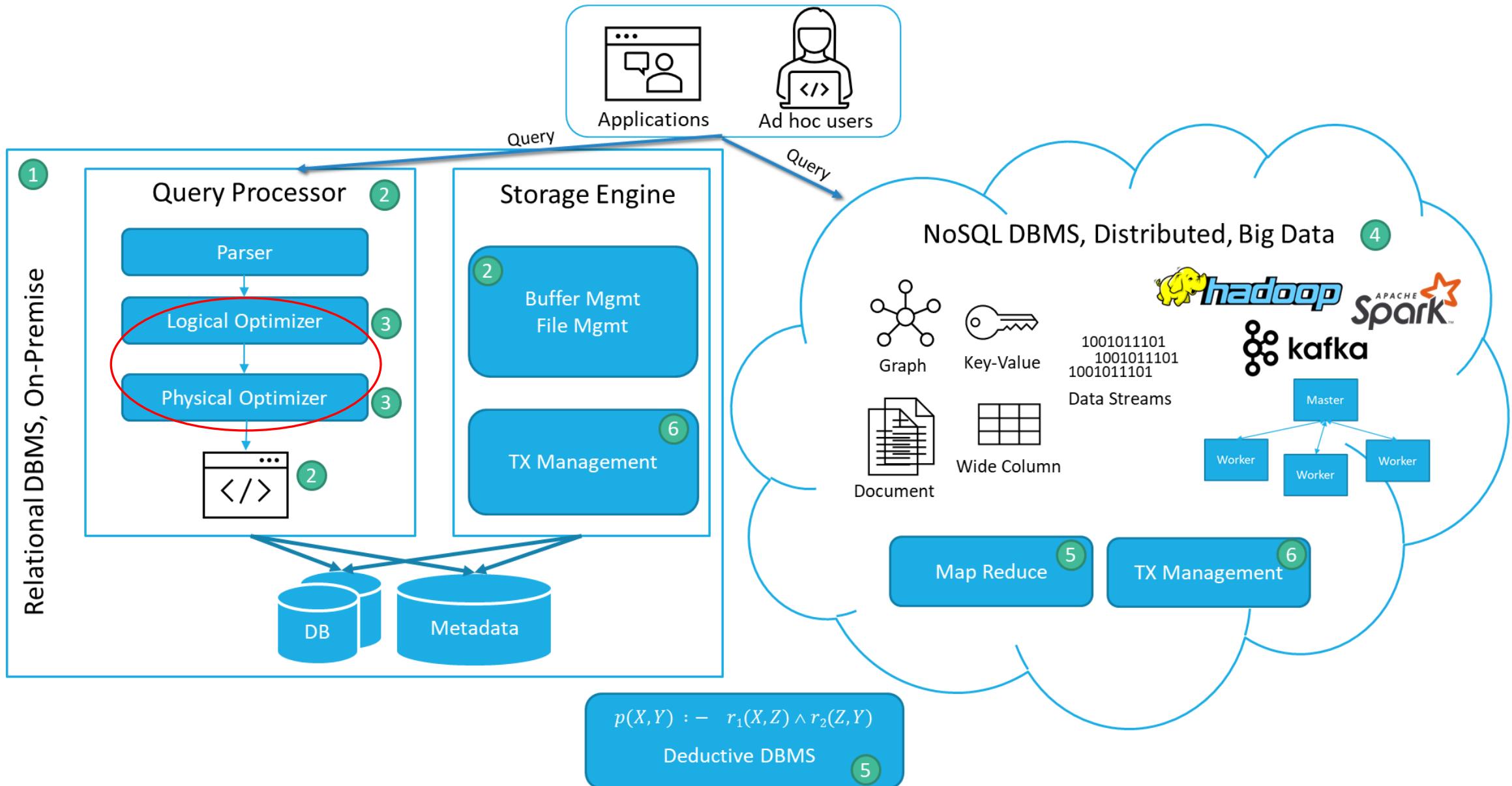
Anastasiia Belova, M.Sc.

Soo-Yon Kim, M.Sc.



Juniorprofessur
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY





3.1 Semantic Query Optimization

Learning Goals

At the end of this section you will be able to

- ✓ know and apply view optimization
- ✓ know the definition of the tableau representation
- ✓ know and apply the tableau method to a given query
- ✓ know and apply a method to test tableau equivalence
- ✓ know and apply method for tableau minimization



Use Semantics to Avoid Joins & Increase Explainability

Faster evaluation as less joins
are used for

- Relational queries
- Deductive queries on fact/rule systems
- Natural language queries (term definitions correspond to rules)

Explanation of
unexpected results

- Violation of integrity constraints (leads to empty results)
- Trivial queries (lead often to very large results)
- Pre-suppositions

Short Excursion - Views



stored query definitions



virtual tables, they do not store any data!



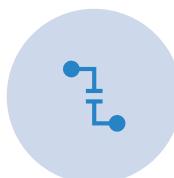
reduce complexity as they usually encapsulate joins between multiple tables



may cover only a subset of the data



pre-aggregate data for analytics using aggregate functions



can be used to control the access to underlying data → define view for a subset & grant access only to that view

Motivating Example – Semantic Optimization of Views

Names of single employees in the Anexa department in a tax bracket under 50%

Views

```
CREATE VIEW AnexaEMP AS  
  
    SELECT e.* FROM EMPL e, DEPT d  
  
    WHERE d.dname= 'Anexa' AND  
          d.dno=e.dno;
```

```
CREATE VIEW TAX50 AS  
  
    SELECT e.* FROM EMPL e  
  
    WHERE (e.marstat='single' AND  
           e.salary<40.000)  
    OR   (e.marstat='married' AND  
           e.salary<80.000);
```

Query

```
SELECT a.name  
  
FROM AnexaEMP a, TAX50 t  
  
WHERE a.marstat= 'single' AND  
      t.eno=a.eno
```

Motivating Example – View Substitution

```
SELECT a.name FROM EMPL a, DEPT d, EMPL t  
  
WHERE d.dname= 'Anexa' AND a.dno=d.dno AND  
      a.marstat='single' AND  
      t.marstat='single' AND  
      t.salary<40.000 AND a.eno=t.eno  
  
OR      d.dname= 'Anexa' AND a.dno=d.dno AND  
      a.marstat='single' AND  
      t.marstat='married' AND  
      t.salary<80.000 AND a.eno=t.eno
```

→ Tableau method for simplification!

Tableau Representation

[Aho et al., 1979;
Abiteboul et al., 1995]

- Representation for a special class of *conjunctive queries* in **domain relational calculus** (DRC)

$$\{a_1 \dots a_m | \exists b_1 \dots b_n (P_1 \wedge P_2 \wedge \dots \wedge P_k)\},$$

where P_i are atomic predicates (relation predicates or comparisons).

- For each relation predicate the tableau contains a row, and for each variable a column
- **Restriction:** attributes are expected to appear only once per relation (*unique role assumption*, no renaming)
- Tableaux correspond to SELECT-PROJECT-JOIN (**SPJ**) queries in Relational Algebra

Construction of tableaux by translating RA expressions into equivalent DRC queries

Tableau Method – Example (1)

```
SELECT a.name FROM EMPL a,  
        DEPT d,  EMPL t  
WHERE  d.dname='Anexa' AND  
        a.dno=d.dno AND  
        a.marstat='single' AND  
        t.marstat='single' AND  
        t.salary<40.000 AND  
        a.eno=t.eno  
  
OR      d.dname='Anexa' AND  
        a.dno=d.dno AND  
        a.marstat='single' AND  
        t.marstat='married' AND  
        t.salary<80.000 AND  
        a.eno=t.eno
```

Equivalent query in Domain Relational Calculus (DRC)

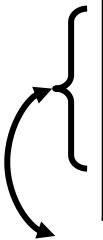
{ n | \exists dname, dno, mst, sal, eno, mgr, n2, mst2, sal2, dno2
EMPL(eno,n,mst,sal,dno) \wedge
DEPT(dno,dname, mgr) \wedge
EMPL(eno,n2,mst2,sal2,dno2) \wedge
dname= ' Anexa' \wedge mst='single' \wedge mst2='single' \wedge
sal2<40.000 } \cup

{ n | \exists dname, dno, mst, sal, eno, mgr, n2, mst2, sal2, dno2
EMPL(eno,n,mst,sal,dno) \wedge DEPT(dno,dname, mgr) \wedge
EMPL(eno,n2,mst2,sal2,dno2) \wedge
dname= ' Anexa' \wedge mst='single' \wedge mst2='married'
 \wedge sal2<80.000}

Tableau Method – Example (2)

{ $n \mid \exists dname, dno, mst, sal, eno, mgr, n2, mst2, sal2, dno2$
EMPL(eno, n, mst, sal, dno) \wedge
DEPT(dno, dname, mgr) \wedge
EMPL(eno, n2, mst2, sal2, dno2) \wedge
dname= 'Anexa' \wedge mst='single' \wedge mst2='single' \wedge sal2<40.000 }

goal row



eno	name	marstat	salary	dno	dname	mgr
	a2					
b1	a2	single	b2	b3	'Anexa'	b4
b1	b5	single	<40.000	b6		

EMPL
DEPT
EMPL

a_i : goal variables
 b_j : bound variables

Syntactic simplification: Removal of superseded rows

Tableau Method – Example (3)

{ n | \exists dname, dno, mst, sal, eno, mgr, n2, mst2, sal2, dno2
EMPL(eno,n,mst,sal,dno) \wedge DEPT(dno,dname, mgr) \wedge
EMPL(eno,n2,mst2,sal2,dno2) \wedge dname= 'Anexa' \wedge
mst='single' \wedge mst2='married' \wedge sal2<80.000}

eno	name	marstat	salary	dno	dname	mgr	
	a2						EMPL
b1	a2	single	b2	b3	'Anexa'	b4	DEPT
b1	b5	married	<80.000	b6			EMPL

Semantic constraint propagation (“chase”) and deletion of contradictory tableaux.

Tableau Method – Example (4)

Result Tableau

eno	name	marstat	salary	dno	dname	mgr	
	a2						
b1	a2	'single'	<40.000	b3	b3	'Anexa'	b4

EMPL
DEPT

{ n | \exists dname, dno, mst, sal, eno, mgr
EMPL(eno,n,mst,sal,dno) \wedge
DEPT(dno,dname, mgr) \wedge
dname= 'Anexa' \wedge mst='single' \wedge sal<40.000 }

```
SELECT a.name FROM EMPL a, DEPT d  
WHERE d.dname= 'Anexa' AND a.dno=d.dno AND  
a.marstat='single' AND  
a.salary<40.000
```

Tableau Containment and Equivalence (1)

- **Optimization Goal:** Find the minimal tableau of all equivalent tableaux for a query.
- **How:** Reduction of tabular rows resulting in reduction of necessary (expensive) joins.

Definition 3.1

Tableau T_1 is *contained* in tableau T_2 ($T_1 \subseteq T_2$) if

1. T_1, T_2 have the same columns and entries in result rows **and**
2. the relation computed from T_1 is a **subset** of the one from T_2 for all valid assignments of relations to rows and for all valid database instances.

- Tableau Containment \Rightarrow Query Containment

Tableau Containment and Equivalence (2)

Theorem 3.1 (Homomorphism Theorem [Abiteboul et al., 1995])

$T_1 \subseteq T_2 \Leftrightarrow$ There is a mapping h from the T_2 symbols to the T_1 symbols with:

1. $h(\text{resulting_row}(T_2)) = \text{resulting_row}(T_1)$
2. $h(\text{row}(T_2)) = \text{any row of } T_1 \text{ with the same relation name}$
3. $h(\text{constant}) = \text{constant}$
4. Integrity constraints in T_2 are transferred to the respective symbols in T_1 and are also guaranteed in T_1 .

Theorem 3.2

Two tableaux T_x and T_y are **equivalent**, denoted as $(T_x \equiv T_y)$

$$\Leftrightarrow T_x \subseteq T_y \wedge T_y \subseteq T_x$$

Example Tableau Containment

$T_1 \subseteq T_2 ? \rightarrow$ Find mapping h from T_2 to T_1
 $T_2 \subseteq T_1 ? \rightarrow$ Find mapping h from T_1 to T_2

T_1	a	
	a	
a	b	(R)
c	d	(R)
e	f	(R)
<hr/>		
	b < d, d < f	

T_2	w	
	w	
w	x	(R)
y	z	(R)
<hr/>		
	x < z	

$T_1 \subseteq T_2$, mapping h exists:

Map row 1 of $T_2 \rightarrow h(w) = a$, $h(x) = b$
Map row 2 of $T_2 \rightarrow h(y) = c$, $h(z) = d$

\rightarrow Check if constraints of T_2 also are valid in T_1
using the mapping: $h(x) < h(z) \rightarrow b < d$ ✓

$T_2 \not\subseteq T_1$, because

Map row 1 of $T_1 \rightarrow h(a) = w$, $h(b) = x$

Map row 2 of $T_2 \rightarrow h(c) = y$, $h(d) = z$

Map row 3 of $T_2 \rightarrow h(e) = y$, $h(f) = z$

Check constraints: $h(b) < h(d) \Leftrightarrow x < z$ ✓
 $h(d) < h(f) \Leftrightarrow z < z$ ⚡

\rightarrow no equivalent for f

Example Tableau Containment

$T_3 \subseteq T_4$? → Find mapping h from T_4 to T_3
 $T_4 \subseteq T_3$? → Find mapping h from T_3 to T_4

T_3	a	b	
	a	c	d (R)
	a	e	f (R)
	g	c	b (R)
	h	e	b (R)

T_4	a	b	
	a	e	f (R)
	h	e	b (R)

Observe, that in T_3 you could remove rows 1 and 3 or 2 and 4 as they are duplicate, as you can set $c=e$ and $g=h$

We remove rows 1 and 3 from T_3

→ $T_3 \subseteq T_4$ and $T_3 \subseteq T_4$ as identity mapping h exists:

- $h(a) = a$
- $h(b) = b$
- $h(e) = e$
- $h(f) = f$
- $h(h) = h$

Tableau Minimization

- For each tableau row: Delete the row and check equivalence to original tableau
- Unfortunately, this minimization is NP-complete (no problem for small tableaux)

Theorem 3.3

Every *minimal tableau* is equivalent to the found tableau (except naming).

- Apply **integrity constraints** (“knowledge-based optimization” using special reasoners, e.g., *chase algorithm*) to find minimal equivalent tableau
- **Key constraints / functional dependencies:** If left sides of FDs (keys) are equal in 2 tableau rows, then right sides are equal as well.
- **Referential constraints:** “pending” rows can be eliminated.
- **Domain constraints:** Constant propagation and elimination of unnecessary comparisons



Quiz

<https://www.menti.com>
Code 3622 4009





3.2 Structure-based Query Optimization

Learning Goals

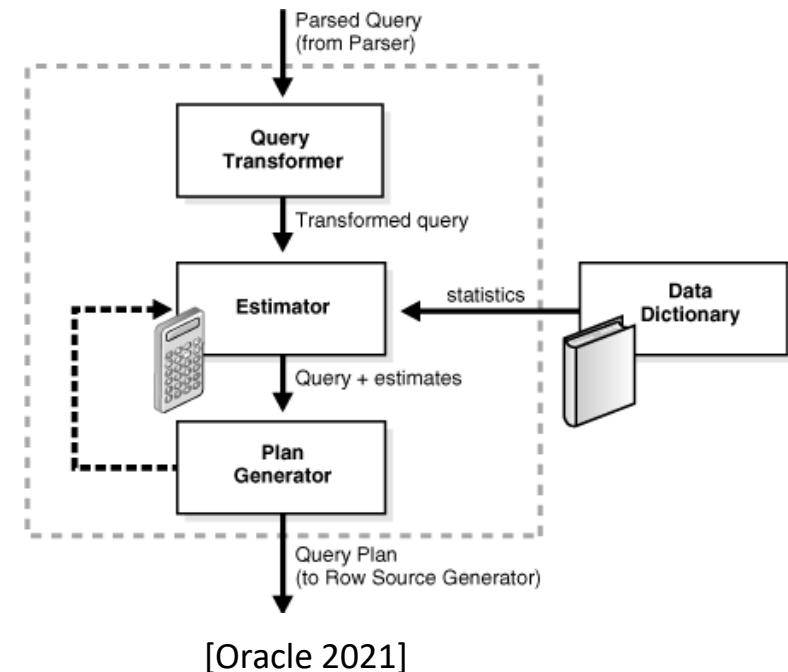
At the end of this section you will be able to

- ✓ create a canonical representation of a RA query
- ✓ draw a syntax tree for an RA expression
- ✓ know RA equivalences and apply corresponding rules to transform an expression
- ✓ name and apply the most important heuristics of an optimizer
- ✓ know and apply subquery optimization



Structure-based QO - Preliminaries

- Based on representation of RA queries as graphs
- Analysis and illustration of structural query properties
- Description of special techniques for query evaluation
- There are various types of query graphs differing in expressiveness
 - **Syntax trees**
 - **Quant graphs** [Jarke & Koch, 1983; Jarke & Koch, 1984]

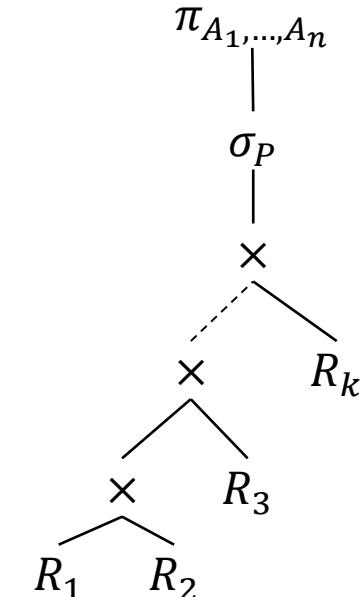


[Oracle 2021]

Logical Query Optimization

- Basis: algebraic normal form
- **Canonical transformation**
Convert query of form `SELECT.. FROM.. WHERE...` into RA expression with cross products, selection, and projection
- **Optimization:** transform into equivalent expressions which are more efficient to execute
- Transformation is based on heuristics
- **Goal:** intermediate results of operators should be as small as possible
- Use tree representation with base relations as leafs

```
SELECT A1, ..., An FROM R1, ..., Rk
WHERE P
```



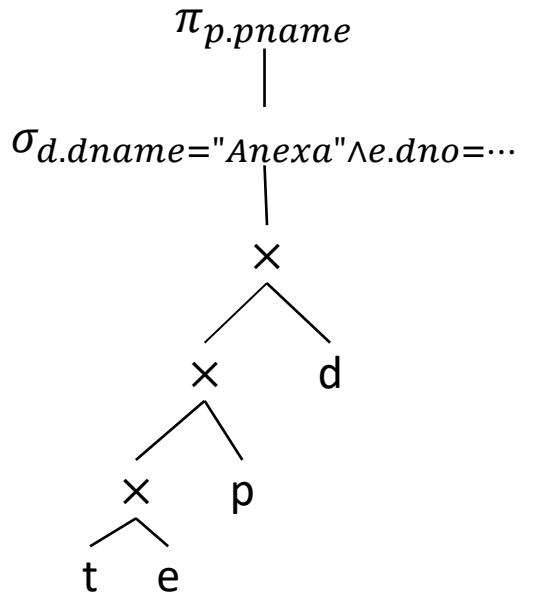
[Eickler & Kemper, 2015]

DEPT	TASK	EMPL	PROJECT
dno	eno	eno	pno
dname	pno	name	pname
mgr	tname	marstat	
	due_date	salary	
		dno	

Amafon Example

List the names of all projects the employees work on in the Anexa department.

```
SELECT DISTINCT p.pname
FROM TASK t, EMPL e, PROJECT p, DEPT d
WHERE d.dname = „Anexa“ AND
      e.dno = d.dno AND
      t.pno = p.pno AND
      t.eno = e.eno
```



Relational Algebra Equivalences

- Let R_1, \dots, R_n be relations, c_1, \dots, c_m be conditions, and a_1, \dots, a_l be attribute sets.

- Selection

- (1) $\sigma_{c_1 \wedge \dots \wedge c_m}(R) = \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_m}(R))\dots))$ nesting or conjunction of conditions
- (2) $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$ change of order

- Projection

- (3) $\Pi_{a_1}(\dots(\Pi_{a_n}(R)\dots)) = \Pi_{a_1}(R)$ eliminate nested projections (given that $a_1 \subseteq \dots \subseteq a_n \subseteq R$)

- Join (union, intersection, and cross product)

- (4) $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$ associative
- (5) $R \bowtie S = S \bowtie R$ commutative

Relational Algebra Equivalences (2)

- A projection commutes with a selection, that only uses attributes retained by the projection.

$$(6) \quad \pi_a(\sigma_c(R)) = \sigma_c(\pi_a(R)), \text{ if } \text{attr}(c) \subseteq a$$

- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.

$$(7) \quad \sigma_{R.X=S.Y}(R \times S) = R \bowtie_{R.X=S.Y} S$$

- A selection just on attributes of R commutes with $R \bowtie S$

$$(8) \quad \sigma_c(R \bowtie S) = \sigma_c(R) \bowtie S$$

- Similarly, if a projection follows a join $R \bowtie S$, we can “push” it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.

$$(9) \quad \pi_a(R_1 \bowtie_c R_2) = \pi_a(\pi_{a_1}(R_1) \bowtie_c \pi_{a_2}(R_2))$$

Relational Algebra Equivalences (3)

- Selection can commute with set operations (union, intersection, and difference), e.g.,

$$(10) \sigma_c(R \cup S) = \sigma_c(R) \cup \sigma_c(S)$$

- Projection can commute with union (but not with intersection and set difference!)

$$(11) \pi_a(R_1 \cup R_2) = \pi_a(R_1) \cup \pi_a(R_2)$$

where R_1 and R_2 have the same schema

- For conditions the usual logical transformations can be used, e.g., using DeMorgan's law

$$(12) \neg(c_1 \vee c_2) = \neg c_1 \wedge \neg c_2$$

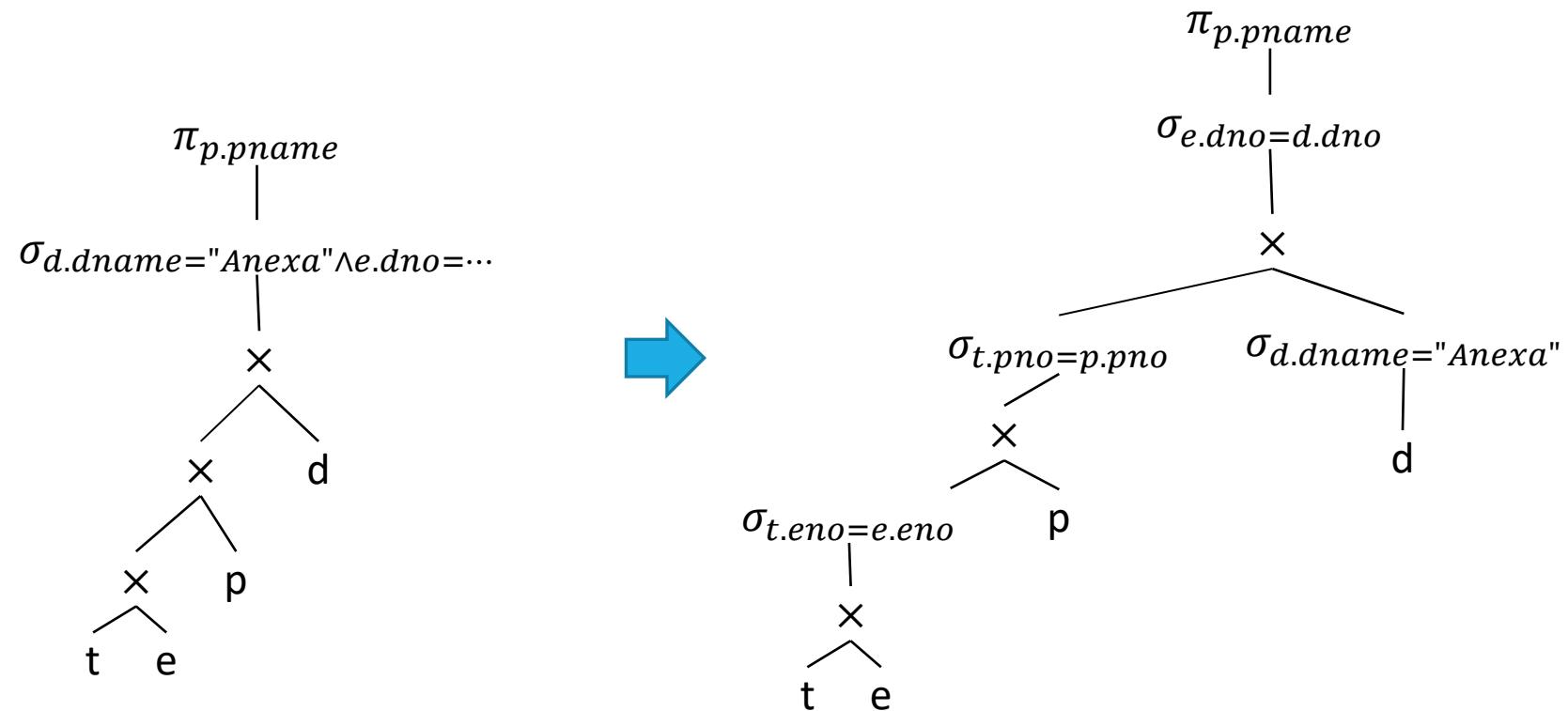
DEPT
<u>dno</u>
dname
mgr

TASK
<u>eno</u>
<u>pno</u>
tname
due_date

EMPL
eno
name
marstat
salary
dno

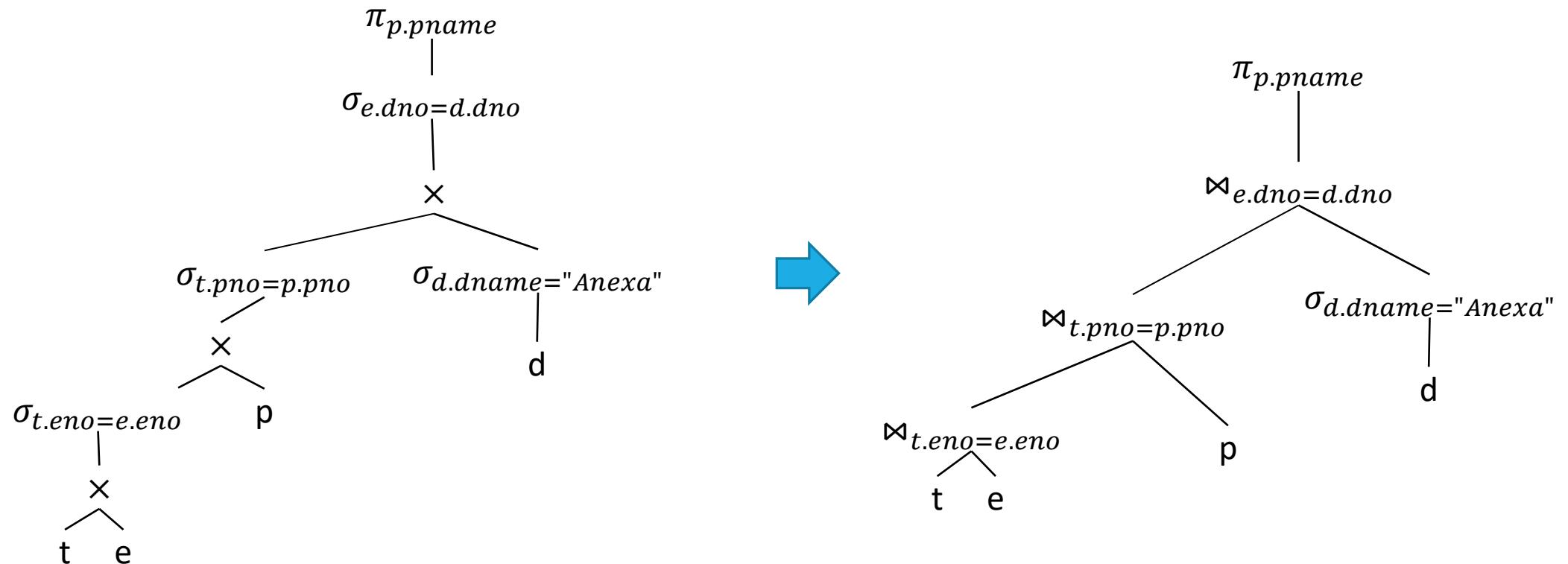
PROJECT

Example – Handle Selections (1)



Break up selections and move them down the tree (Rule 1)

Example – Handle Selections (2)



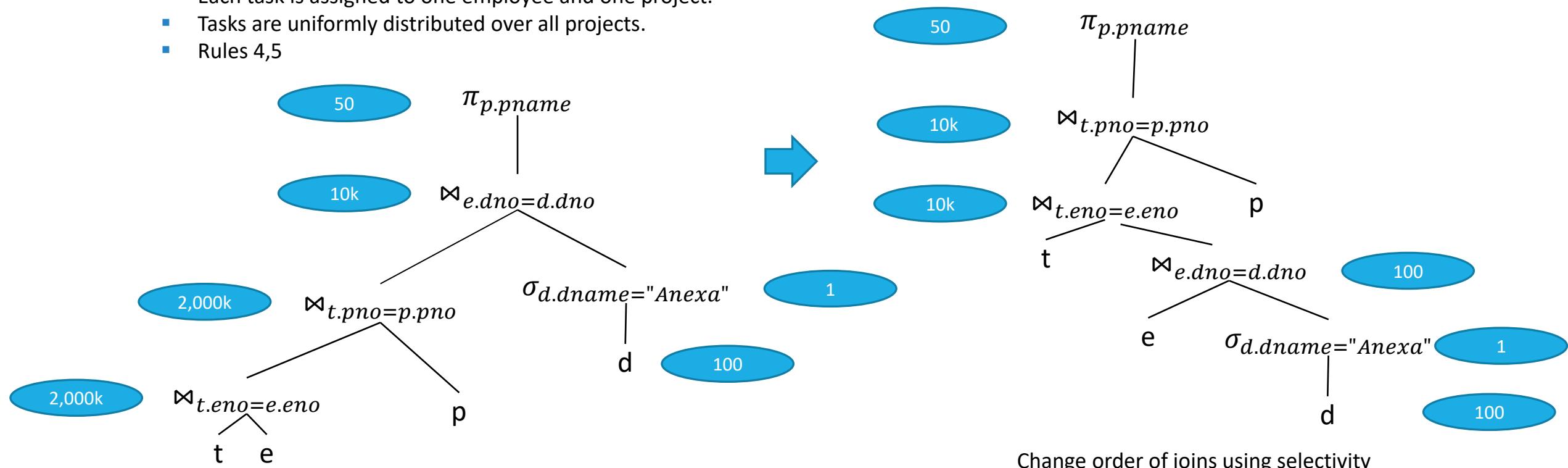
Merge cross products and selections (Rule 7)

Example – Join Order

DEPT	TASK	EMPL	PROJECT
dno	eno	eno	pno
dname	pno	name	pname
mgr	tname	marstat	
	due_date	salary	
		dno	

100 2,000k 40k

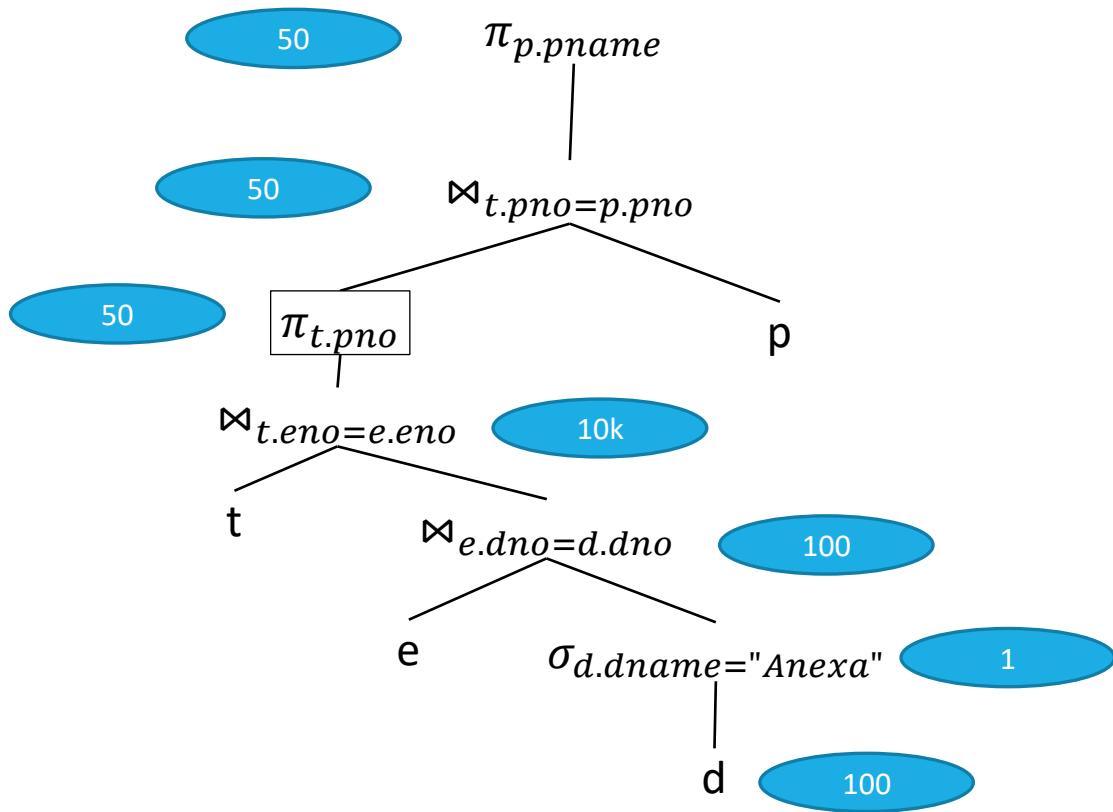
- In the Anexa department there are 100 employees which are involved in 50 projects.
- Each task is assigned to one employee and one project.
- Tasks are uniformly distributed over all projects.
- Rules 4,5



Example – Add and Move Projections

DEPT	TASK	EMPL	PROJECT
dno	eno	eno	pno
dname	pno	name	pname
mgr	tname	marstat	
	due_date	salary	
		dno	
			10k
			40k

- Rules: 3, 6, 9, 11
- Trade-off between duplicate elimination costs and join costs
- **Heuristics:**
 - If value range of the projected attributes is small in comparison to the number of attributes **or** if the values of the removed attribute are big (e.g., media files), it is beneficial to use it
- Also possible with other operators, e.g., aggregations



Heuristics Summary

Basis: Canonical normal form

1. Break up selections
2. Move selections as far as possible down the tree
3. Merge cross-products and selections to joins
4. Determine order of joins based on selectivity → get as small intermediate results as possible
5. Optionally add projections
6. Move projections as far as possible down the tree



Nested Subquery Optimization

- **Nested subquery:** Query inside the WHERE block
- **Correlated subquery:** inner query contains a reference to the outer query
- Naïve strategy: evaluate the nested inner query for each tuple of the outer relation
→ very inefficient

```
SELECT e1.name, e1.salary  
FROM EMPL e1  
WHERE e1.salary = (SELECT MAX (e2.salary) FROM EMPL e2 )
```

Nested Subquery Optimization (2)

- Wherever possible, optimizer tries to convert queries with nested subqueries into a join
- EXISTS: Subquery works like a function returning true or false (is department name „Anexa“?)

```
SELECT e.name, e.salary  
FROM EMPL e  
WHERE EXISTS (  
    SELECT *  
    FROM DEPT d  
    WHERE  
        d.dno = e.eno AND  
        d.name = „Anexa“)
```



```
SELECT e.name, e.salary  
FROM EMPL e, DEPT d  
WHERE d.dno = e.dno AND  
      d.dname = „Anexa“
```

Nested Subquery Optimization (3)

- In general queries with involving a nested subquery connected by IN or ANY can always be converted into a single block query
- Further option → transformation using **semi-join** (non-standard notation $R_1.a \text{ } S= R_2.b$)

```
SELECT d.dname  
FROM DEPT d  
WHERE d.dno IN  
(SELECT e.dno FROM EMPL e WHERE e.salary < 40,000)
```



```
SELECT d.dname  
FROM DEPT d, EMPL e  
WHERE d.dno S= e.eno AND  
e.salary < 40,000
```

- NOT EXISTS, NOT IN, ALL subqueries can be resolved with the **anti-join** ($R_1.a \text{ } A= R_2.b$)
- Anti-join semantics: tuple of R1 is rejected as soon as $R_1.a$ finds a match with any value of $R_2.b$ → tuple of R_1 is only returned, if there is no match in R_2

```
SELECT COUNT(*)  
FROM EMPL e  
WHERE e.dno NOT IN  
(SELECT d.dno FROM DEPT d WHERE d.dname „Anexa“)
```



```
SELECT COUNT(*)  
FROM DEPT d, EMPL e  
WHERE e.dno A= d.dno AND  
d.dname = „Anexa“
```



Quiz

<https://www.menti.com>
Code 3622 4009





3.3 Cost-based Query Optimization

Learning Goals

At the end of this section you will be able to

- ✓ describe the components of a cost-based query optimizer
- ✓ explain the most important steps in Selinger-style cost-based query optimization
- ✓ describe selectivity factors and calculate them
- ✓ know different types of histograms and when to use them
- ✓ determine the costs for single and multiple relation query plans
- ✓ describe dynamic programming in QO and the optimality criterion



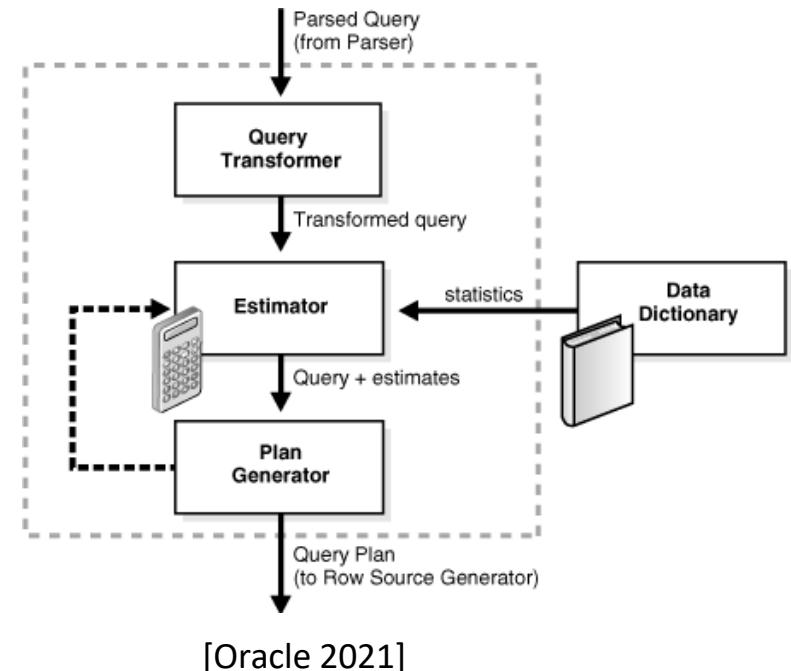
Optimization of a Relational Query

- Semantic Query Optimization
- Structure-based Query Optimization
- **Cost-based Query Optimization**
 - Based on statistics collected about the data to be accessed, e.g., relation sizes, access paths
 - Create alternative query plans
 - Block-wise planning (query block = single SELECT-FROM-WHERE, no nesting)
→ query subplans

Cost-based Query Optimization

First step: create alternative plans

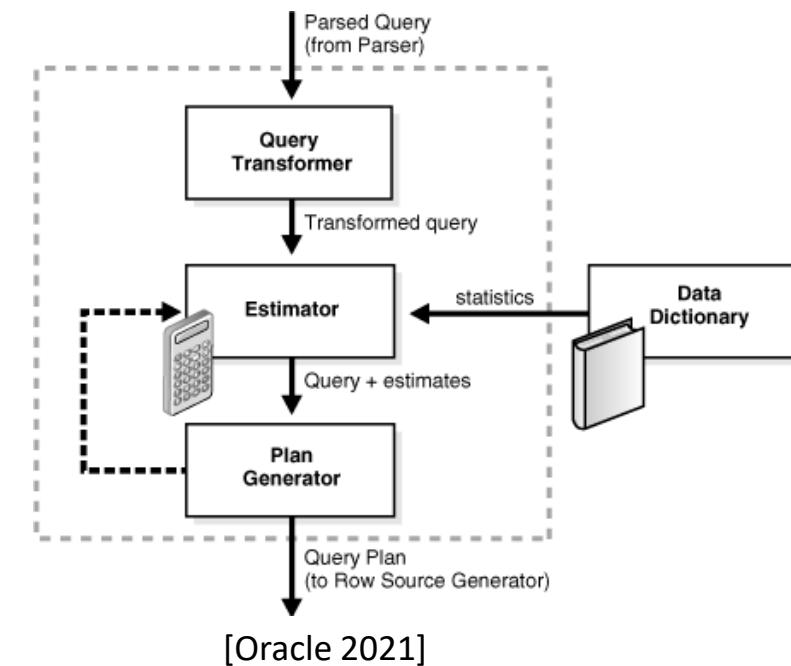
- Access methods, e.g., full table scan or index scans
- Operator implementations, e.g., join methods
- Join orders etc.
- Algebraic equivalence transformations
→ exponential no. of equivalent expressions
(e.g., $n!$ for n joins)



Cost-based Query Optimization (Selinger-style / System-R-Style)

Next: Use estimation model (cost model)

- Search solution space to a problem for a solution minimizing an *objective (cost) function*
 - Needs to **estimate** the costs of an operator (or a sequence of operators) and the size of its result
 - Not exact → may select not the optimal one



Query Optimization in System R (IBM 1979), still the foundation for Query Optimization in DBMS today [Selinger et al., 1979]

Costs in Query Optimization

[Elmasri & Navathe 2017]

Access cost to secondary storage	Disk storage cost	Computation cost (CPU costs/time)	Memory usage cost	Communication cost
<ul style="list-style-type: none">Affected by indexes, sorting, type of access structures etc. (see chapter 2)Main emphasis for big DBsSimple cost functions ignore other factors	<ul style="list-style-type: none">Cost for storing intermediate files created by an execution strategy	<ul style="list-style-type: none">Performing in-memory operations in data buffers during query execution (e.g., sorting, merging)Interesting for small DBs, in-memory DBs	<ul style="list-style-type: none">Costs for main memory buffers needed during query execution	<ul style="list-style-type: none">Costs for shipping query & results between DB and clientInteresting for distributed DBs to transfer tables, subqueries, results etc. among nodes

Difficult to consider all cost types in a (weighted) function → how should we define the weights?

```
SELECT name, salary  
FROM EMPL e  
WHERE salary < 40,000  
AND marstat = 'single'
```

Cost Estimation: Query with Single Relation

- Basic cost formula

secondary storage
accesses (Page Fetches)

$$\text{COST} = \text{PF} + w \cdot (\text{predicted_#tuples})$$

weighting factor between I/O
and CPU time

- Statistical data

- For each relation R
 - $B(R)$: number of pages needed to store relation R
 - $T(R)$: number of tuples of relation R → **Cardinality**

- For each attribute a of R
 - $V(R,a)$: number of distinct values relation R has in attribute a

- **Selectivity factor (F)**: corresponds to expected fraction of tuples fulfilling a condition on an attribute

[Selinger et al., 1979;
Garcia-Molina et al., 2009]

```

SELECT name, salary
FROM EMPL e
WHERE salary > 40,000
AND marstat = 'single'

```

Estimation of Selectivity Factors

Query cardinality: cardinalities of relations in FROM list * product of selectivity factors of filter predicates.

Condition	Selectivity Factor F	Remarks
col=val	$1/V(R,col)$	1/10 if there is no index
col1=col2	$1/\max(V(R,col1),V(R,col2))$	$1/V(R,col_i)$ if there is only an index on col i, 1/10 if there is no index
col>val	$(\text{highkey} - \text{val}) / (\text{highkey} - \text{lowkey})$	1/3 if col. is not arithmetic or no index
col BETWEEN val1 AND val2	$(\text{val2}-\text{val1}) / (\text{highkey} - \text{lowkey})$	1/4 if col. is not arithmetic
col IN (list of vals)	$\min(\text{list} * F(\text{col}=\text{value}), \frac{1}{2})$	
col IN subquery	$(\text{estimated cardinality of subquery result}) / (\prod_{R_i \text{ IN FROM-List of subquery}} T(R_i))$	
pred1 OR pred2	$F(\text{pred1}) + F(\text{pred2}) - F(\text{pred1}) * F(\text{pred2})$	
pred1 AND pred2	$F(\text{pred1}) * F(\text{pred2})$	Assumption: Column values must be independent
NOT pred	$1 - F(\text{pred})$	

Note: There might be multiple ways to compute the selectivity for a predicate!

[Selinger et al., 1979; Elmasri & Navathe, 2017]

Example – Selectivity Factor

EMPL
<u>eno</u>
name
marstat
salary
dno

- $T(EMPL) = 40k$
- $V(EMPL, sal) = 80$
- Highest= 100,000
- Lowest = 10,000
- $V(EMPL, marstat) = 4$
- Both have an index

Q1:

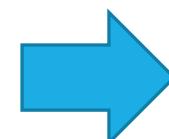
```
SELECT name, salary  
FROM EMPL e  
WHERE salary > 40,000 AND marstat = 'single'
```

Q2:

```
SELECT name, salary  
FROM EMPL e  
WHERE salary > 40,000 OR marstat = 'single'
```

$$F(sal > 40,000) = \frac{100,000 - 40,000}{100,000 - 10,000} = \frac{2}{3}$$

$$F(marstat = 'single') = \frac{1}{4}$$

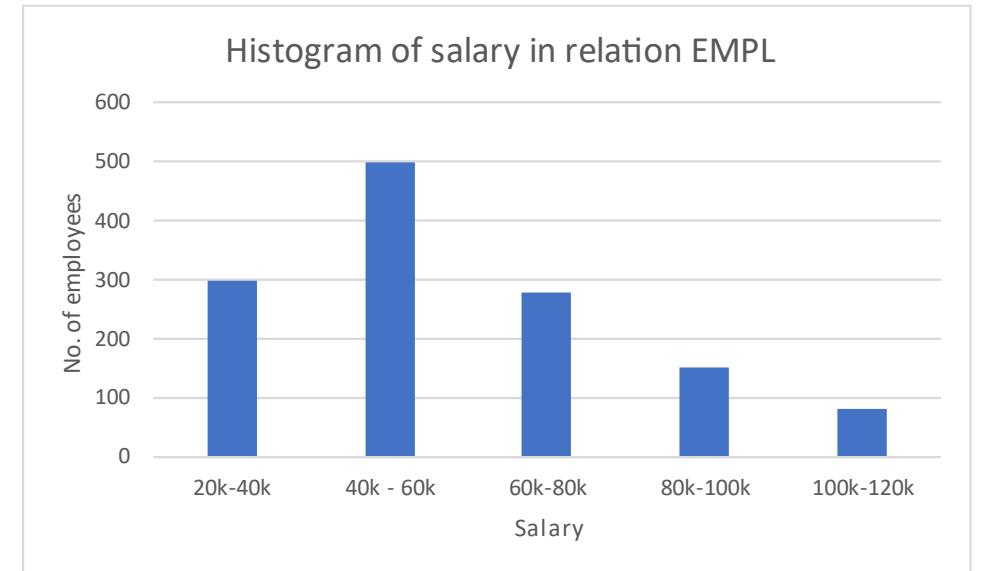


$$T(Q_1) = 40,000 \cdot \frac{2}{3} \cdot \frac{1}{4} = \frac{40,000}{6} \approx 6,667$$

$$T(Q_2) = 40,000 \cdot \left(\frac{2}{3} + \frac{1}{4} - \left(\frac{2}{3} \cdot \frac{1}{4} \right) \right) = \frac{120,000}{4} = 30,000$$

Histograms - More accurate selectivity estimation

- Data structure to approximate data distribution
 - Range of values is divided into subranges (**buckets**)
 - In each bucket, number of tuples with a value in that range is counted
1. **Equi-width histogram:** Subranges have equal sizes
 2. **Equi-height histogram:** Number of tuples in each range is equal (or at least similar)
 3. **V-optimal histogram:**
 - Minimize variance of frequency approximation
 - Buckets for “outliers”
- **Advantage:** Optimization sensitive to available statistics
 - **Disadvantage:** Expensive to collect and maintain





Quiz

<https://www.menti.com>
Code 3622 4009



Dynamic Programming



Goal: Find the optimal plan for
 $\text{Join}(R_1, \dots, R_n)$

For each S in $\{R_1, \dots, R_n\}$ do:
Find optimal plan for $\text{Join}(\text{Join}(\{R_1, \dots, R_n\} \setminus S), S)$
Pick the plan with the lowest total cost



Principle of Optimality

Optimal plan for a larger expression requires
optimal plans for its sub-expressions



Complexity

Enumeration cost drops from $O(n!)$ to $O(n \cdot 2^n)$
May need to store $O(2^n)$ partial plans
Significantly more efficient than the naïve scheme

Bottom-up approach: start with the single relations!

Single-relation Query Plans

- **Goal:** find plans with minimal costs and/or interesting properties
- Interesting properties:
 - Order specified in query in ORDER BY or GROUP BY clause
 - Orders which might be beneficial for a subsequent operators (e.g., joins)
- **Output**
 - Access path costs: scan, indexes
 - Order: either via index or other sorting produced
- Access paths considered further
 - Cheapest path with some “interesting order”
 - The cheapest unordered plan with any other order

→ If no “interesting order” is given, only cheapest plan is of interest

Costs for Single-relation Query Plans

Context	Cost
Unique index for equal predicate	$\log_G 0.15B(R) + w \cdot 1$ Note: Only one result tuple!
Applicable* clustered index	$\log_G 0.15B(R) + F(pred) \cdot B(R) + w \cdot \#tuples$
Applicable non-clustered index	$\log_G 0.15B(R) + F(pred) \cdot T(R) + w \cdot \#tuples$
Scan of Relation	$B(R) + w \cdot T(R)$

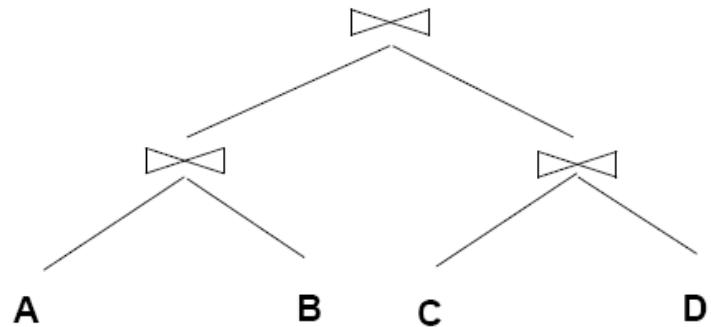
* **Applicable:** Index I contains column of one or more conjunct in WHERE clause of conjunctive query

[Selinger et al., 1979]
See also slide 48 in Chapter 2

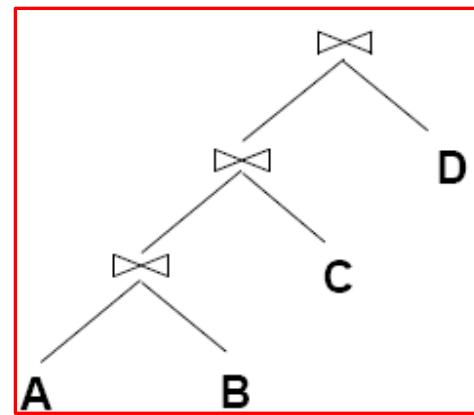
Cost Estimation and Join Strategies

- Methods
 - Check all applicable join methods
 - Check combination of join methods for n-pass joins
- Costs of join evaluation depend on sequence of joins, but the final result does not
- Final result can be estimated by relation cardinalities & selectivity factors
- Examination of all possible sequences of joins (exponential!) is avoided by heuristics
 - Don't consider join sequences which involve Cartesian products or do them as late as possible
 - Consider only a specific form of join order: left-deep plans

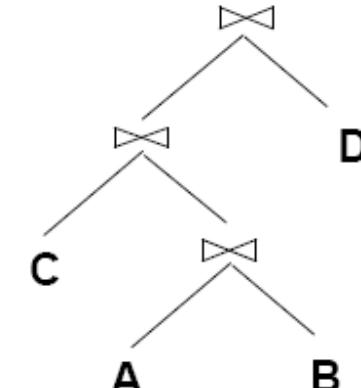
Left-Deep Plans



Bushy



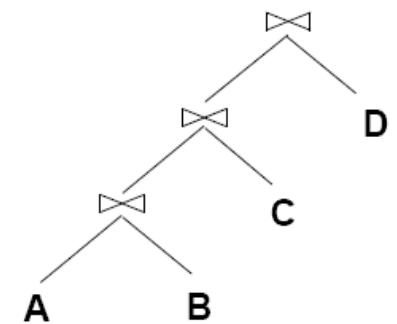
Left-Deep
(right is a base table)



Linear
(at least one child
is a base table)

Optimizers only consider left-deep plans

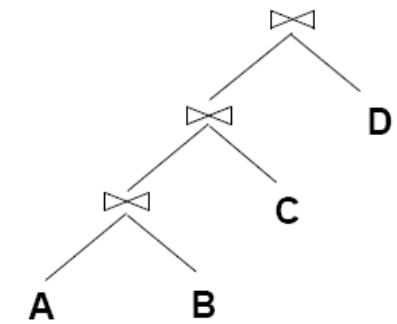
- Smaller solution space → may not find the cheapest plan, but is more efficient
- Supports pipelining of joins → do not need to materialize join result



Enumeration of Left-Deep Plans (1)

- **Left-deep plans differ only in**
 - order of relations
 - access method for each relation
 - join method for each join
 - **Enumerated using N passes (if N relations joined)**
 - **Pass 1:** Find best 1-relation plan for each relation.
 - **Pass 2:** Find best way to join result of each 1-relation plan (as outer) to another relation.
(All 2-relation plans)
 - **Pass N:** Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation.
(All N-relation plans)
 - **For each subset of relations, retain only**
 - Cheapest plan overall, plus
 - Cheapest plan for each *interesting order* of the tuples.

Enumeration of Left-Deep Plans (2)



- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an *interestingly ordered* plan or an additional sorting operator.
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, unless all predicates in WHERE have been used up
 - Avoid Cartesian products if possible
- Approach still requires exponential plan space in the number of tables
 - **Note:** This shows the need for semantic optimization. It is more efficient to avoid a join than to optimize its execution.



Quiz

<https://www.menti.com>
Code 3622 4009



Example

Based on [Selinger et al., 1979]

```
SELECT e.name, t.tname, e.sal, d.dname
FROM EMP e, DEPT d, TASK t
WHERE t.tname='Design'
AND d.dname='ANEXA'
AND e.dno=d.dno
AND e.eno=t.eno
```

DEPT	TASK	EMPL	PROJECT
<u>dno</u>	<u>eno</u>	<u>eno</u>	<u>pno</u>
lname	pno	name	pname
mgr	tname	marstat	
	due_date	salary	
		dno	

DEPT	dno	dname	mgr
50	ANEXA	5	
51	BILL	6	
52	SHIP	7	

EMPL	eno	name	dno	sal
3	Smith	50	85	
4	Jones	50	150	
5	Doe	51	95	

TASK	eno	tname	pno
3	Design	34	
4	Req.	33	
5	Impl.	34	
3	Design	46	

1 Relevant single-relation query plans

- each “interesting” order
- Other orders (incl. no order) if access is cheaper than cheapest “interesting” order

Example: Query Plans for Single Relations

- Scan of each relation
 - EMPL (sorted by eno)
 - DEPT (sorted by dno)
 - TASK (sorted by eno)
- Indexes
 - EMPL.dno
 - EMPL.eno (clustered)
 - DEPT.dname
 - TASK.eno (clustered)
 - TASK.tname
- Required information
 - Costs (I/O + CPU, but we will just consider I/O costs in the following)
 - Size of result (after selection): #tuples and #pages
 - Order (if any)

Example: Costs for Single Relation Query Plans

Query plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	B(EMPL)	T(EMPL)	B(EMPL)	eno
EMPL.dno	-	0.15*B(EMPL)+T(EMPL)	T(EMPL)	B(EMPL)	dno
EMPL.eno	-	(0.15+1.5)* B(EMPL)	T(EMPL)	B(EMPL)	eno
Scan DEPT	dname= ,Anexa'	B(DEPT)	$F_D * T(DEPT)$	$F_D * B(DEPT)$	dno
DEPT.dname	dname=,Anexa'	0.15* B(DEPT) + $F_D * T(DEPT)$	$F_D * T(DEPT)$	$F_D * B(DEPT)$	-
Scan TASK	tname='Design'	B(TASK)	$F_T * T(TASK)$	$F_T * B(TASK)$	eno
TASK.eno	tname='Design'	(0.15+1.5)* B(TASK)	$F_T * T(TASK)$	$F_T * B(TASK)$	eno
TASK.tname	tname='Design'	0.15*B(TASK)+ $F_T * T(TASK)$	$F_T * T(TASK)$	$F_T * B(TASK)$	-

F_D and F_T are the selectivity factors for the selection on TASK and DEPT

Example: Insert Concrete Values

Worst case estimation, most likely cheaper in practice

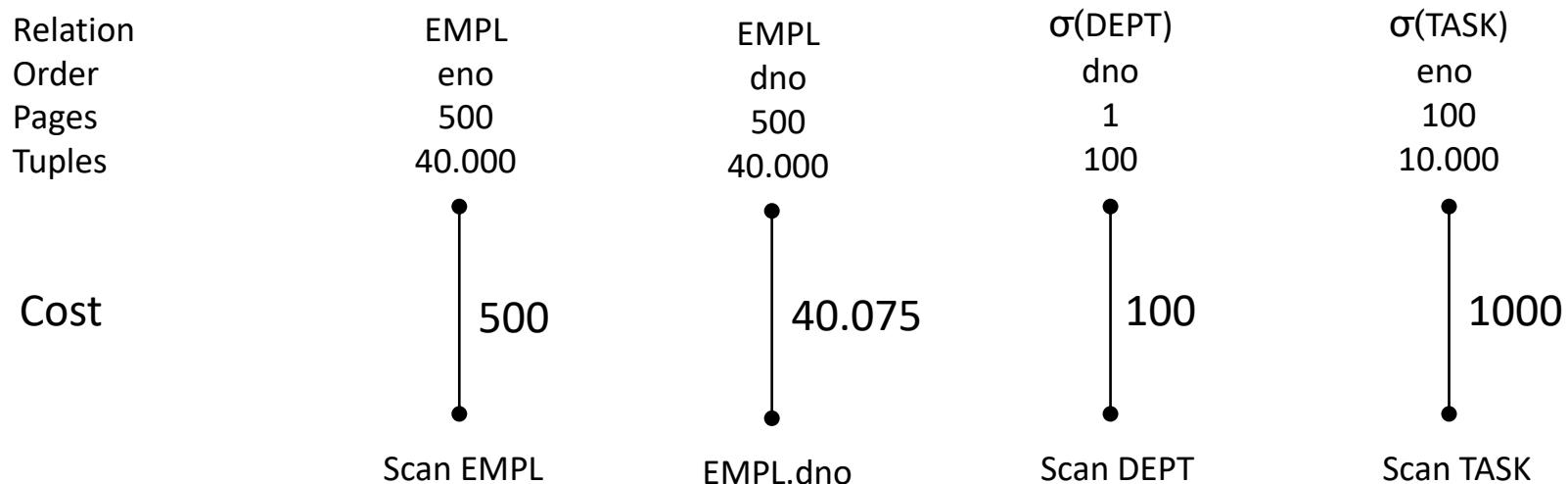
Access plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	500	40.000	500	eno
EMPL.dno	-	40.075	40.000	500	dno
EMPL.eno	-	825	40.000	500	eno
Scan DEPT	dname='Anexa'	100	100	1	dno
DEPT.dname	dname='Anexa'	115	100	1	-
Scan TASK	tname='Design'	1.000	10.000	100	eno
TASK.eno	tname='Design'	1.650	10.000	100	eno
TASK.tname	tname='Design'	10.150	10.000	100	-

$$\begin{aligned}B(\text{EMPL}) &= 500 \\T(\text{EMPL}) &= 40.000 \\ \\B(\text{TASK}) &= 1000 \\T(\text{TASK}) &= 100.000 \\F_T &= 0.1 \\ \\B(\text{DEPT}) &= 100 \\T(\text{DEPT}) &= 10.000 \\F_D &= 0.01\end{aligned}$$

Example: Remove Non-optimal Subplans

Access plan	Selection	Costs	#tuples	#pages	Order
Scan EMPL	-	500	40.000	500	eno
EMPL.dno	-	40.075	40.000	500	dno
EMPL.eno	-	825	40.000	500	eno
Scan DEPT	dname='Anexa'	100	100	1	dno
DEPT.dname	dname='Anexa'	115	100	1	-
Scan TASK	tname='Design'	1.000	10.000	100	eno
TASK.eno	tname='Design'	1.650	10.000	100	eno
TASK.tname	tname='Design'	10.150	10.000	100	-

Example: Remaining Plans



Example: Two-Relation plans

2

Compute costs of all relevant join strategies for all remaining pairs of relations (avoiding cross products). Find the best results for

- $\text{EMPL} \bowtie \text{TASK}$
- $\text{EMPL} \bowtie \text{DEPT}$

$$X = \frac{\# \text{tuples} * \text{tuple size}}{\text{page size}}$$

Access plan	BNL	Merge	#tuples	#pages	Order
Scan EMPL \bowtie Scan TASK	$B(E) + B(E)*B(T')/(B-2)$	$B(E) + B(T')$	$F_{ET} * T(E) * T(T')$	X	eno (Merge)
EMPL.dno \bowtie Scan TASK	$B(E) + B(E)*B(T')/(B-2)$	$B(E) + B(T') + B(E)\log B(E)$	$F_{ET} * T(E) * T(T')$	X	eno (Merge)
Scan EMPL \bowtie Scan DEPT	$B(D') + B(D')*B(E)/(B-2)$	$B(E) + B(D') + B(E)\log B(E)$	$F_{ED} * T(E) * T(D')$	X	dno (Merge)
EMPL.dno \bowtie Scan DEPT	$B(D') + B(D')*B(E)/(B-2)$	$B(E) + B(D')$	$F_{ED} * T(E) * T(D')$	X	dno (Merge)

F_{ET} and F_{ED} are the selectivity factors for joins $\text{EMPL} \bowtie \text{TASK}$ and $\text{EMPL} \bowtie \text{DEPT}$

T' and D' refer to the relations TASK and DEPT after selection

Example: Two-Relation plans with values

$$F_{ET}=1/40.000 \quad F_{ED}=1/100 \quad \text{Buffer}=52$$

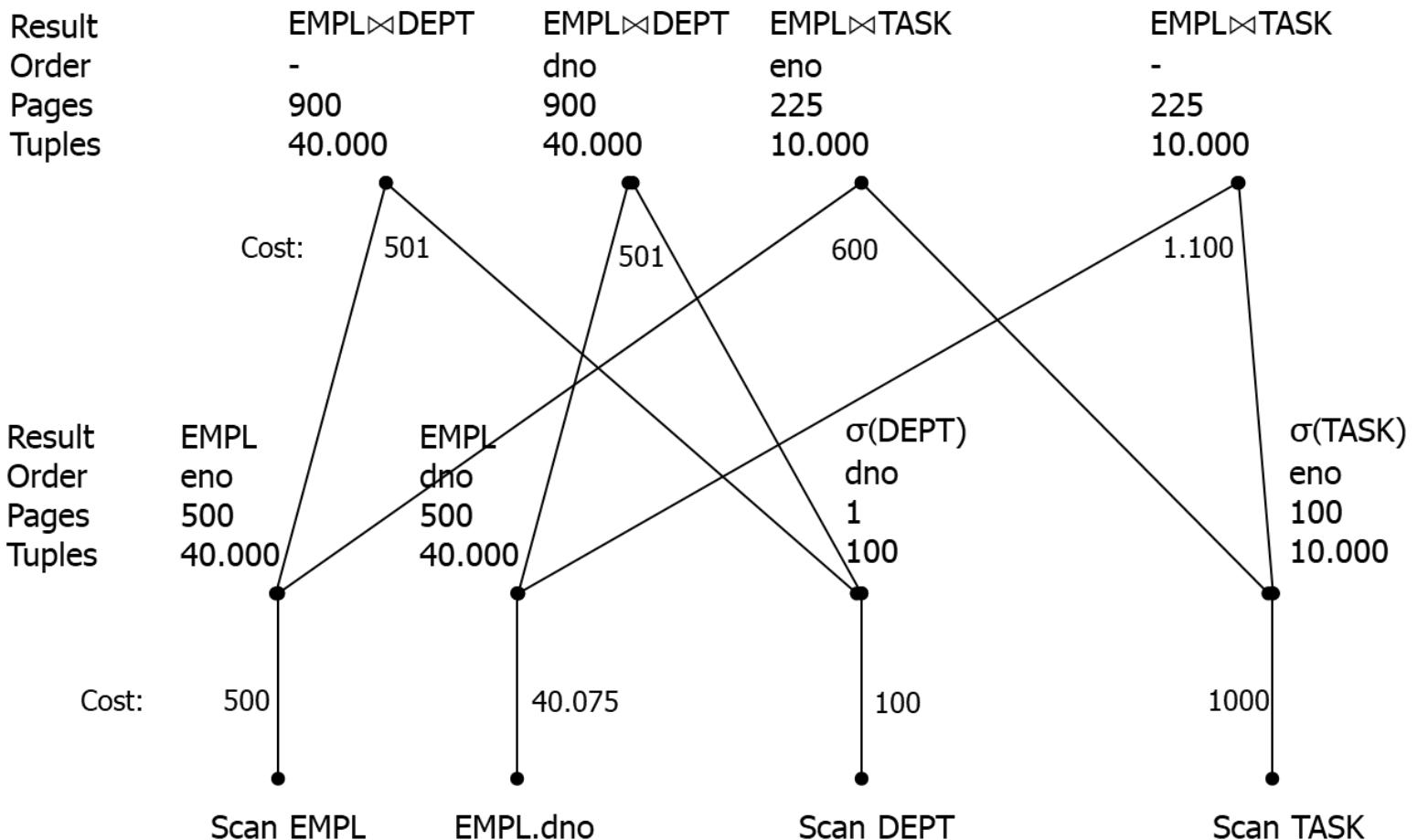
Assumption: Sorting for Merge-Join requires only one additional pass

Access plan	BNL	Merge	#tuples	#pages	Order of Best	Sub-plan	Best Total
Scan EMPL \bowtie Scan TASK	1.500 ₍₁₎	<u>600</u>	10.000	225	eno	1.500	2.100
EMPL.dno \bowtie Scan TASK	<u>1.500₍₁₎</u>	1.600	10.000	225	-	41.075	42.675
Scan EMPL \bowtie Scan DEPT	<u>501</u>	1.501	40.000	900	-	600	1.101
EMPL.dno \bowtie Scan DEPT	<u>501</u>	<u>501</u>	40.000	900	dno	40.175	40.676

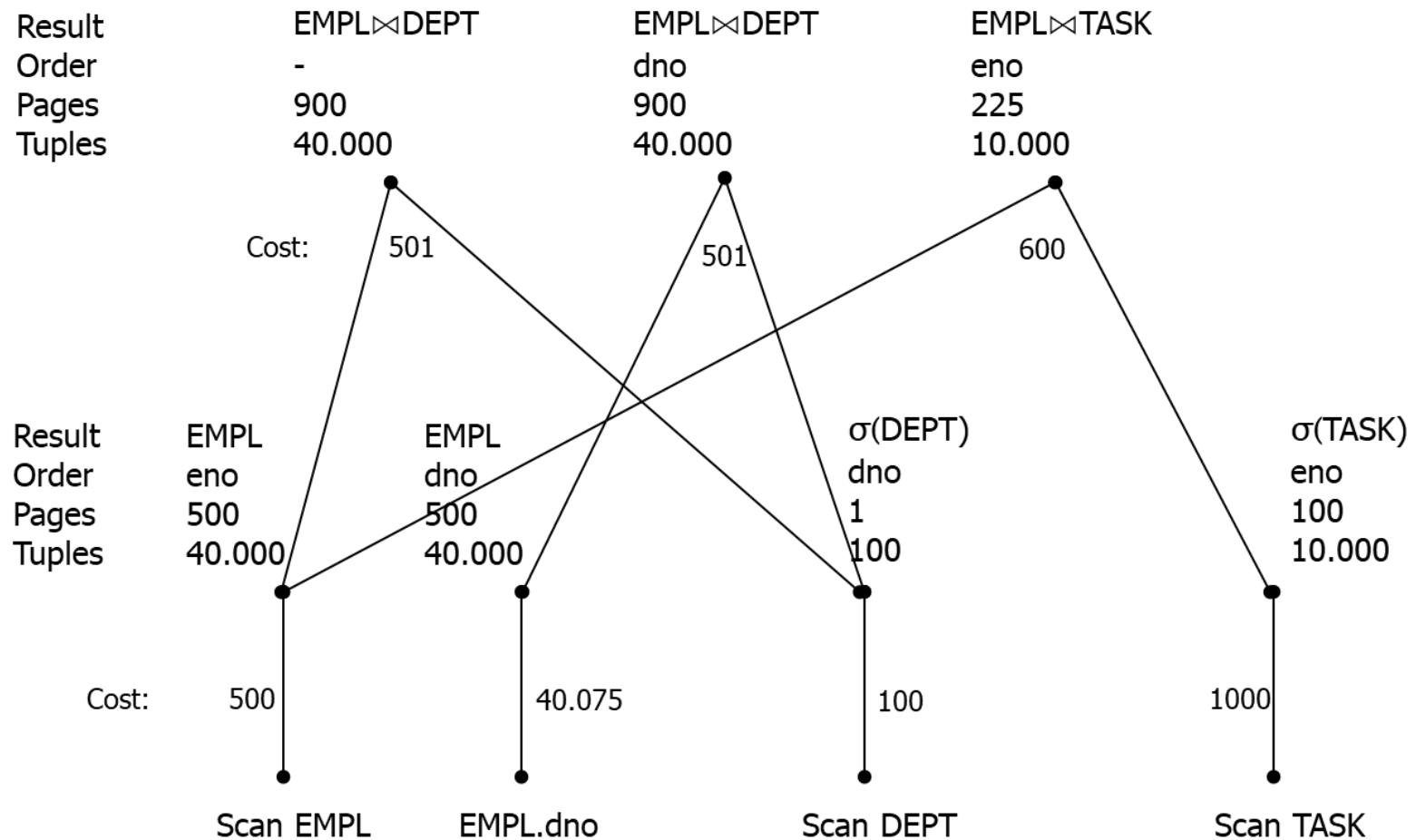
(1) Is 1100 if TASK is the outer relation

Plan with BNL join can be removed, because it does not preserve any order and is more expensive than the first plan. Plan with merge join preserves eno order as the first plan, but is also more expensive than the first. Therefore, it will be also removed.

Example: Graph for 2- Relation Plans



Example: Graph for Remaining 2- Relation Plans



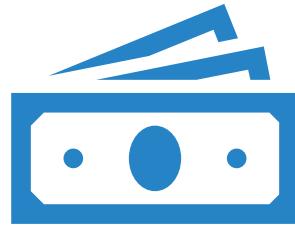
Example: 3-Relation Plans

Access plan	BNL	Merge	Sub-plan	Best Total
(Scan EMPL \bowtie_M Scan TASK) \bowtie Scan DEPT	<u>226</u>	676	2.100	2.326
(Scan EMPL \bowtie_B Scan DEPT) \bowtie Scan TASK	<u>1.900</u>	2.800	1.101	3.001
(EMPL.dno \bowtie_M Scan DEPT) \bowtie Scan TASK	<u>1.900</u>	2.800	40.676	42.576

And the winner is ...

Merge join for joining EMPL and TASK (using relation scans), and then BNL join with DEPT

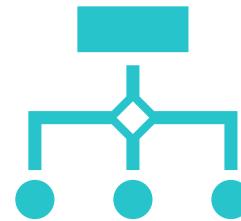
Summary: System R Approach



Cost model based on

Access methods

Size and cardinality of relations



Enumeration exploits

Dynamic programming

One optimal plan for each equivalent expression
(taking into account interesting orders)

Limitations of System R Approach

Cost Model

- One aggregate number for every column (inaccurate)
- Independence assumption

Transformation

- Limited to join ordering

Enumeration

- Limited to single block queries

Query Optimization in Today's Systems

Projection

- Hash-based and sort-based implementations

Joins

- PNL, INL, BNL, Hash, Sort-Merge

Estimation of Query Costs

- Histograms (equi-depth with some variations)
- Sampling for building histograms

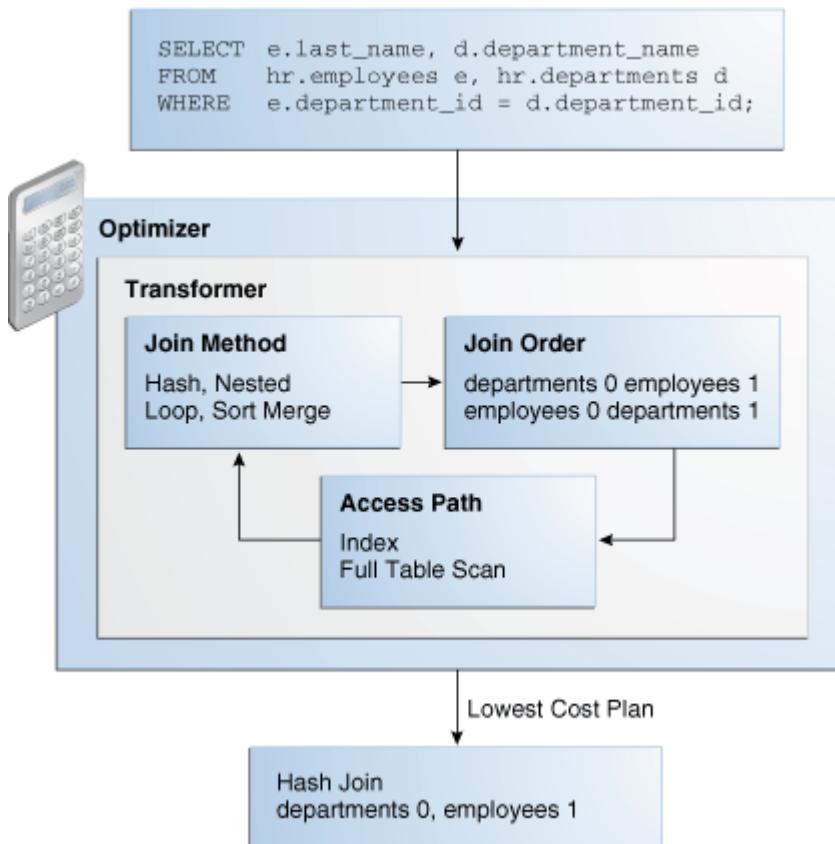
Index-Only Queries

- DBMS try to build index-only plans
- Recommend columns to be included in index, but not part of key

Query Optimization

- Dynamic Programming with left-deep plans with some variations
- User might control optimizer, e.g., define join order or edit query execution plan
- Adaptive query optimization → run-time adjustments to execution plans, discover additional information that can lead to better statistics
- Approximate query processing: optimization techniques for analytic queries calculating results with a certain error range, e.g., for BI tasks on large data sets

Example: Oracle Query Optimizer



GENERAL PLANS

Considering cardinality-based initial join order.
Permutations for Starting Table :0

Join order[1]: DEPARTMENTS[D]#0 EMPLOYEES[E]#1

Now joining: EMPLOYEES[E]#1

NL Join

Outer table: Card: 27.00 Cost: 2.01 Resp: 2.01 Degree: 1
Bytes: 16

Access path analysis for EMPLOYEES

...

Best NL cost: 13.17

...

SM Join

SM cost: 6.08
resc: 6.08 resc_io: 4.00 resc_cpu: 2501688
resp: 6.08 resp_io: 4.00 resp_cpu: 2501688

...

SM Join (with index on outer)
Access Path: index (FullScan)

...

HA Join

HA cost: 4.57
resc: 4.57 resc_io: 4.00 resc_cpu: 678154
resp: 4.57 resp_io: 4.00 resp_cpu: 678154

Best:: JoinMethod: Hash
Cost: 4.57 Degree: 1 Resp: 4.57 Card: 106.00 Bytes: 27

...

Join order[2]: EMPLOYEES[E]#1 DEPARTMENTS[D]#0

...

Now joining: DEPARTMENTS[D]#0

...

HA Join

HA cost: 4.58
resc: 4.58 resc_io: 4.00 resc_cpu: 690054
resp: 4.58 resp_io: 4.00 resp_cpu: 690054

Join order aborted: cost > best plan cost

Best:: JoinMethod: Hash
Cost: 4.57 Degree: 1 Resp: 4.57 Card: 106.00 Bytes: 27

[Oracle 2021]

Review Questions

- Which types of query optimization did we consider?
- What is the difference between semantic and structure-based query optimization?
- What is the goal of the tableau method?
- What is the principle of optimality in dynamic programming? How is it applied in query optimization?
- Which strategies are applied to reduce the number of query plans to be considered in cost-based query optimization?
- Which sub-plans will be maintained for the next stage in cost-based query optimization?
- Why do we need selectivity estimation? What is a histogram?

References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

[Abiteboul et al., 1995] Abiteboul, S.; Hull, R.B.; Vianu, V.: Foundations of Databases. Addison-Wesley, 1995.
<http://webdam.inria.fr/Alice/>

[Aho et al., 1979] Aho, A. V.; Sagiv, Y. & Ullman, J. D.: Efficient optimization of a class of relational expressions. ACM Transactions on Database Systems (TODS), ACM, 1979, 4, 435-454.

[Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). Fundamentals of database systems (Vol. 7). Pearson

[Jarke & Koch, 1983] Jarke, M. & Koch, J.: Range nesting: A fast method to evaluate quantified queries. ACM SIGMOD Record, 1983, 13, 196-206.

[Jarke & Koch, 1984] Jarke, M. & Koch, J.: Query optimization in database systems. ACM Computing surveys (CsUR), ACM, 1984, 16, 111-152.

[Kemper & Eickler, 2015] Kemper, A., & Eickler, A. (2015). Datenbanksysteme; 10., akt. u. erw. Aufl. (in German)

[Selinger et al., 1979] Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A. & Price, T. G.: Access path selection in a relational database management system. Proceedings of the 1979 ACM SIGMOD international conference on Management of data, 1979, 23-34.

[Oracle 2021] Oracle SQL Tuning Guide – Query Optimizer Concepts, <https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/query-optimizer-concepts.html>

Implementation of Databases

Chapter 4: Systems and Architectures
for Big Data

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

Excercises

Anatasiia Belova, M.Sc.

Soo-Yon Kim, M.Sc.



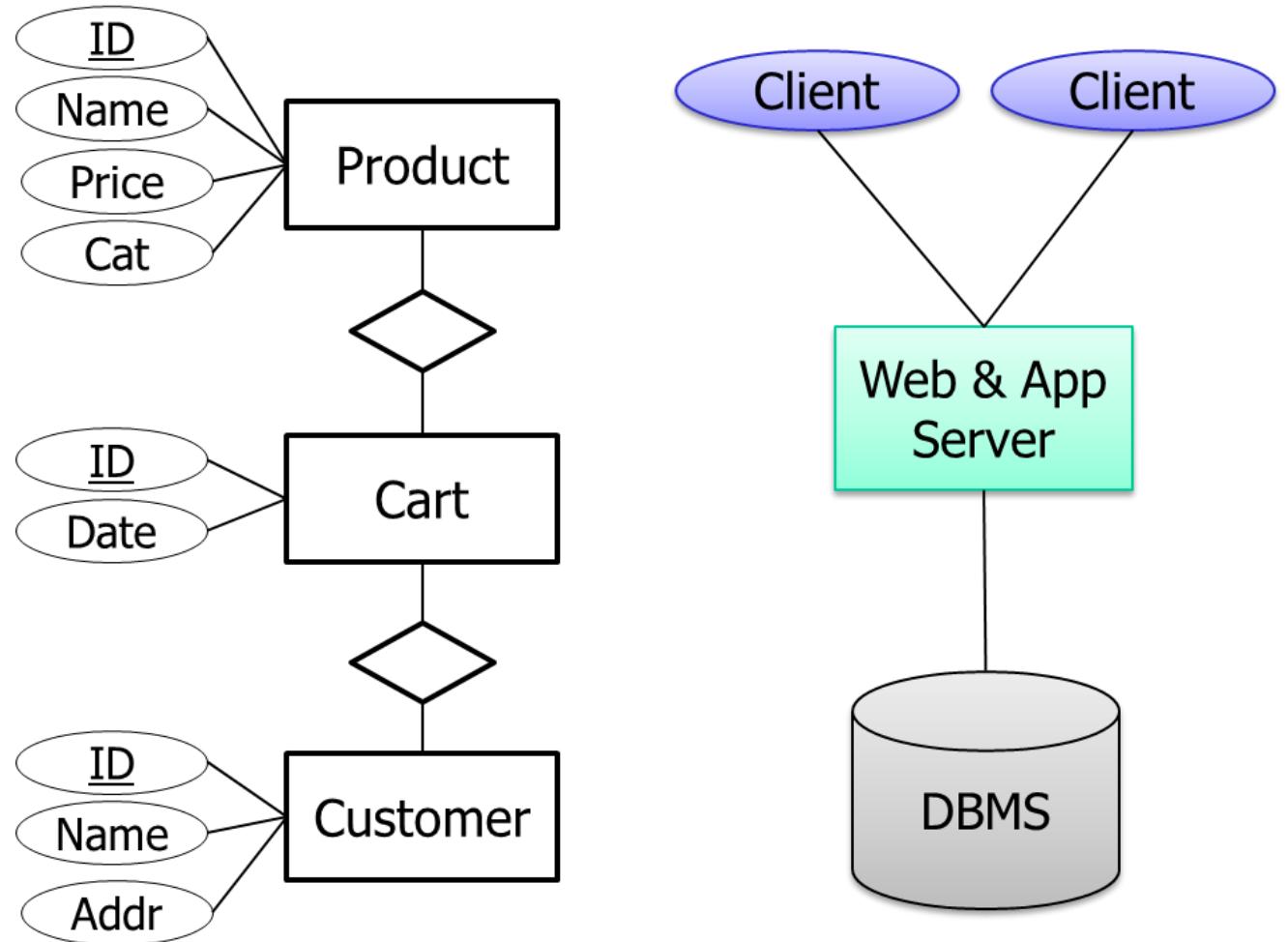
Juniorprofessur
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY

Review of RDBMS Technology Advantages

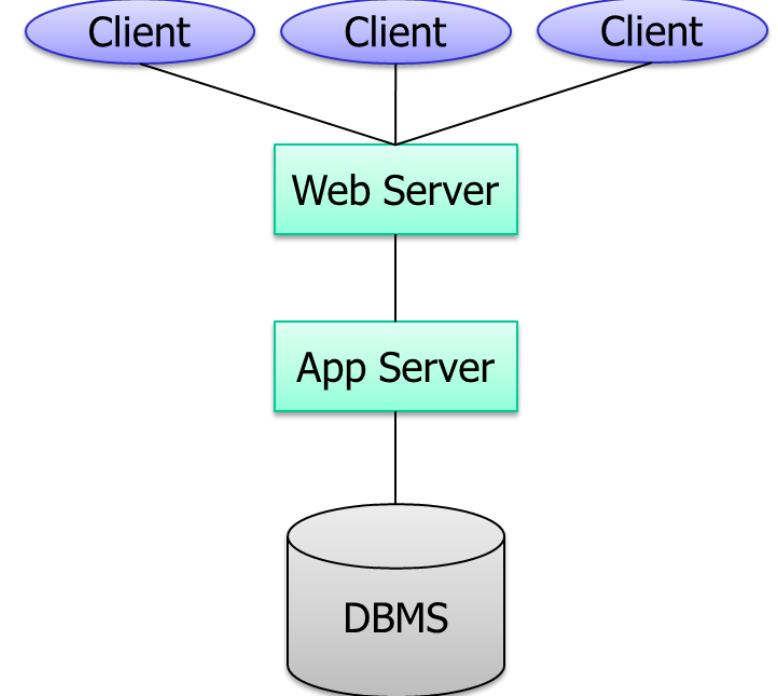
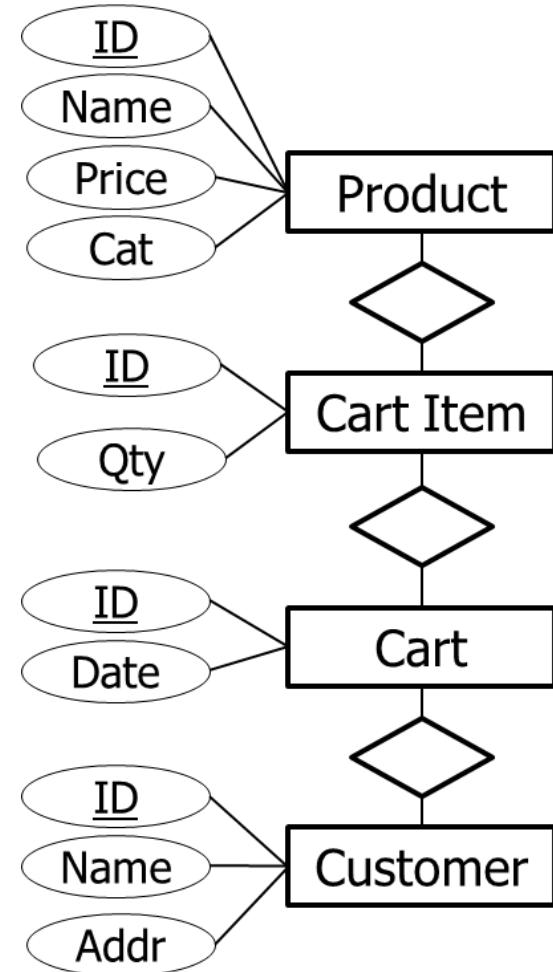
Simple & efficient data model	Expressive query & update language	Powerful transaction management	Basis for application integration
<ul style="list-style-type: none">• Relations, tuples, keys, foreign keys, ...• Standardized for more than 20 years	<ul style="list-style-type: none">• Enables complex queries with nesting, aggregates, functions, ...• Updates can be based on query results	<ul style="list-style-type: none">• Enables concurrent access by multiple users & applications• ACID principle guarantees: Atomicity, Consistency, Isolation, and Durability of TXs	<ul style="list-style-type: none">• Applications use the same shared database instance

A Typical Internet Application



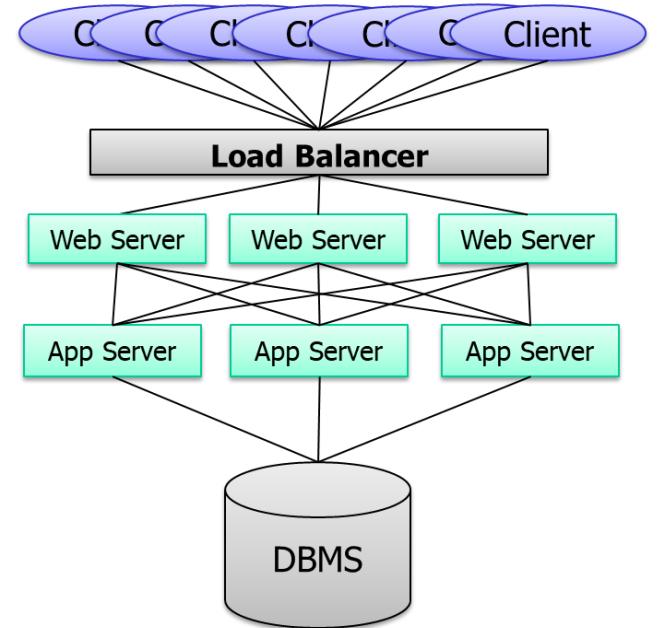
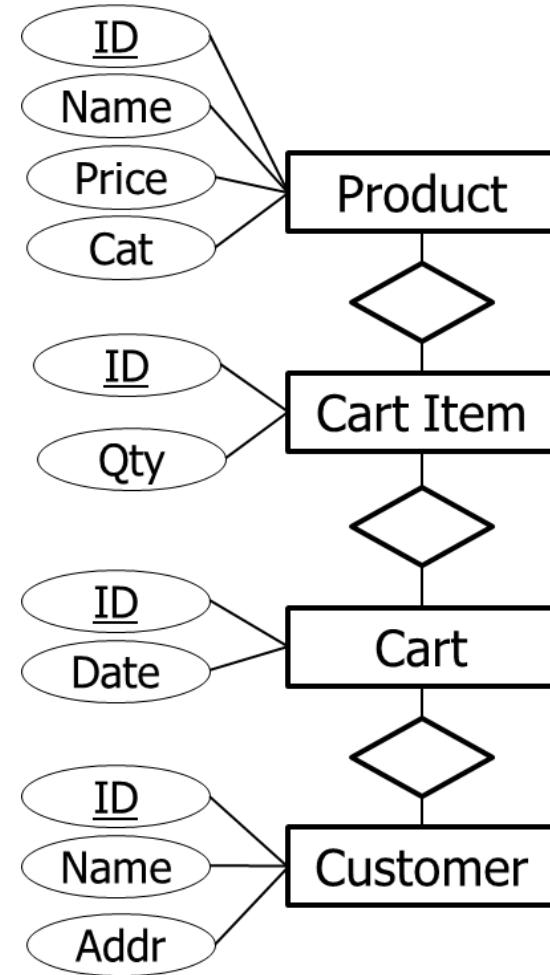
Web shop with some products, customers, carts, etc.

A Typical Internet Application



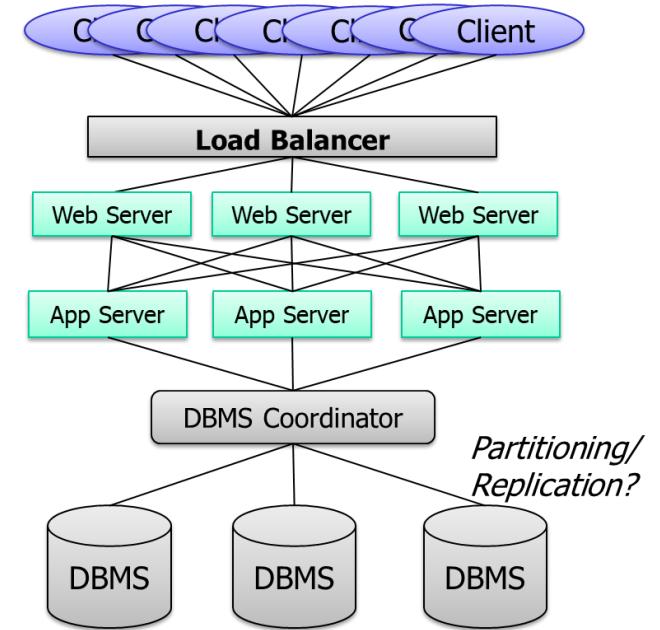
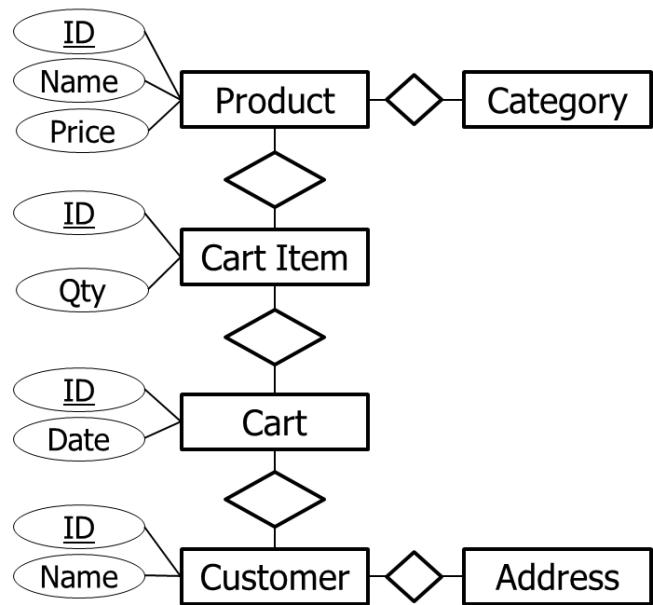
Business is going well, more customers visit your site ...

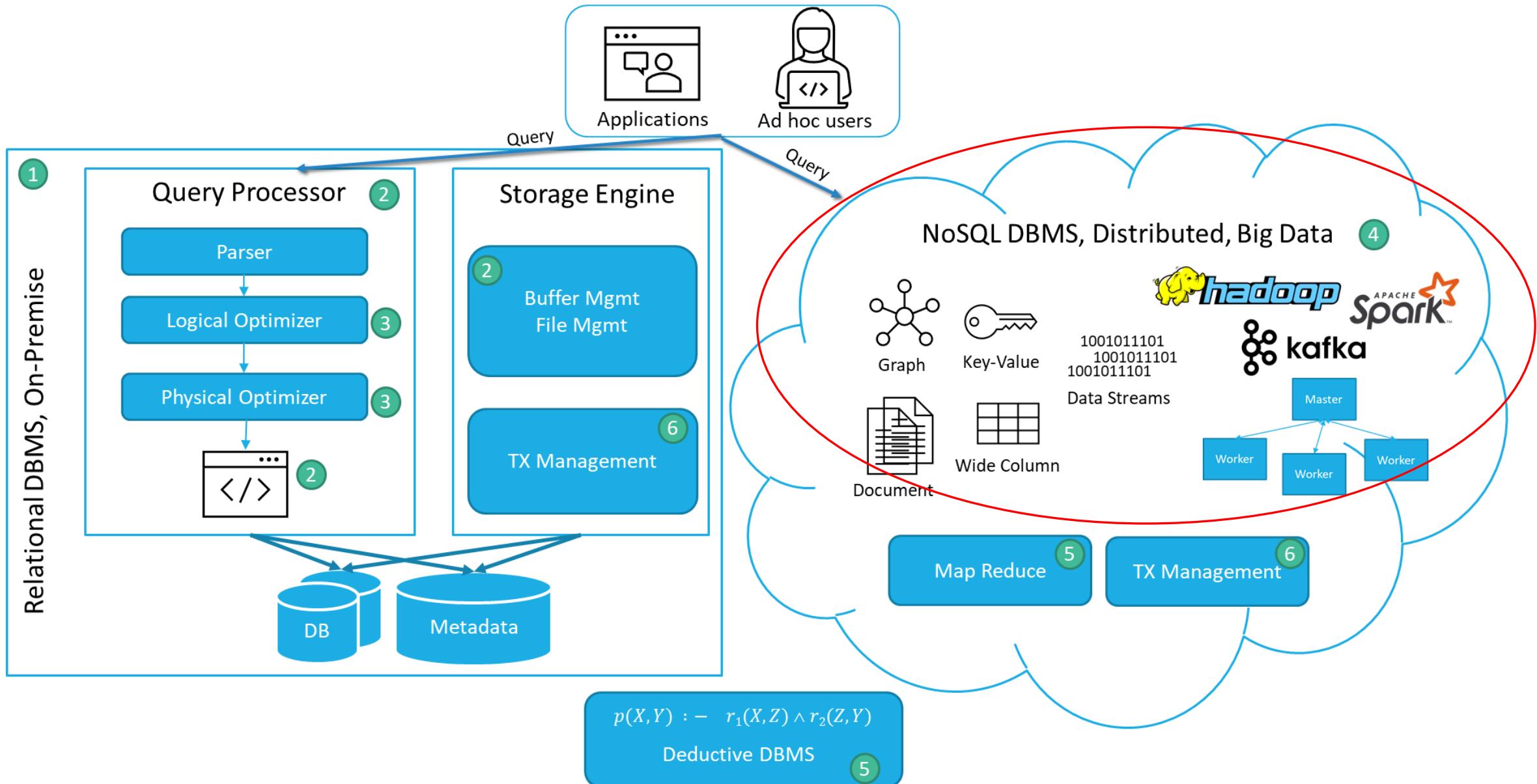
A Typical Internet Application



Business is going *very well*, many customers visit your site ...

A Typical Internet Application







4.1 Data Management in the Cloud

Learning Goals

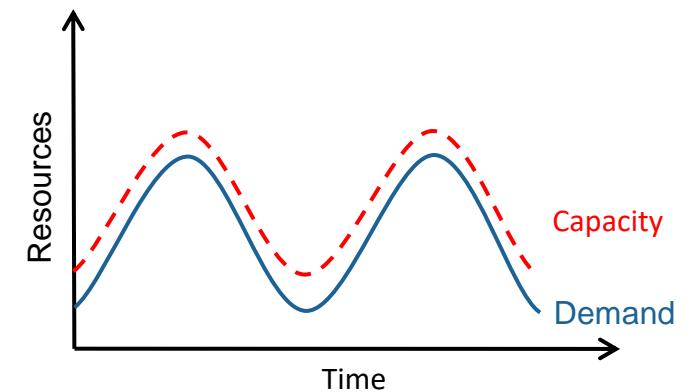
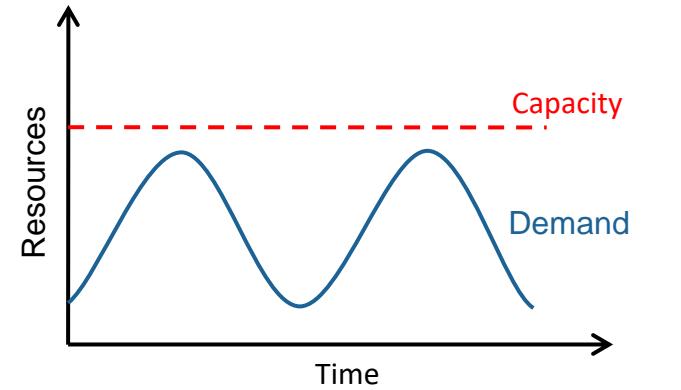
At the end of this section you will be able to

- ✓ know and explain the disadvantages of RDBMS for huge web applications
- ✓ explain the CAP theorem and categorize systems into AP,CA,CP
- ✓ explain and compare consistency models in distributed databases
- ✓ know and explain characteristics of NoSQL systems in general
- ✓ explain the properties of the four main NoSQL DB types and know an example
- ✓ know and apply design choices for document-oriented databases
- ✓ explain the creation and querying of Neo4J graphs
- ✓ explain sharding



Requirements

- High performance, availability, elasticity, heterogeneity
- Distribution
 - Distribute data to multiple nodes to handle workload
 - Process queries in parallel to increase performance
 - Update handling requires communication between nodes to keep consistency
- Scalability
 - Elasticity: Add / remove new resources with changing demand
 - RDBMS are not well designed for distribution
 - Parallel RDBMS are expensive



[Agrawal et al., 2011]

Problems with RDBMS: Data Model and Query Language



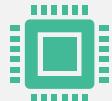
Effort for database design & schema definition is too high

A-priori definition of schema is not possible with frequently changing requirements
Relational model does not fit application model



Many applications require only simple retrieve & update queries

Joins are required due to normalization in RDBMS
Result tuples cannot be updated directly

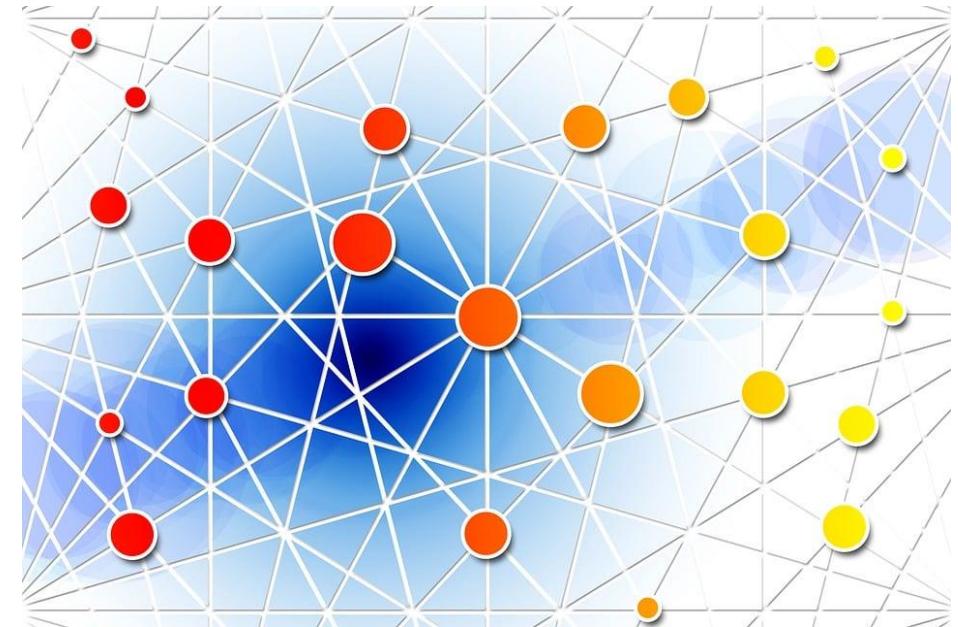


Application integration done by service-oriented architecture

Application database vs. shared database
Services are used for application integration rather than direct access to underlying databases

Problems with RDBMS: Transaction Management and Scalability

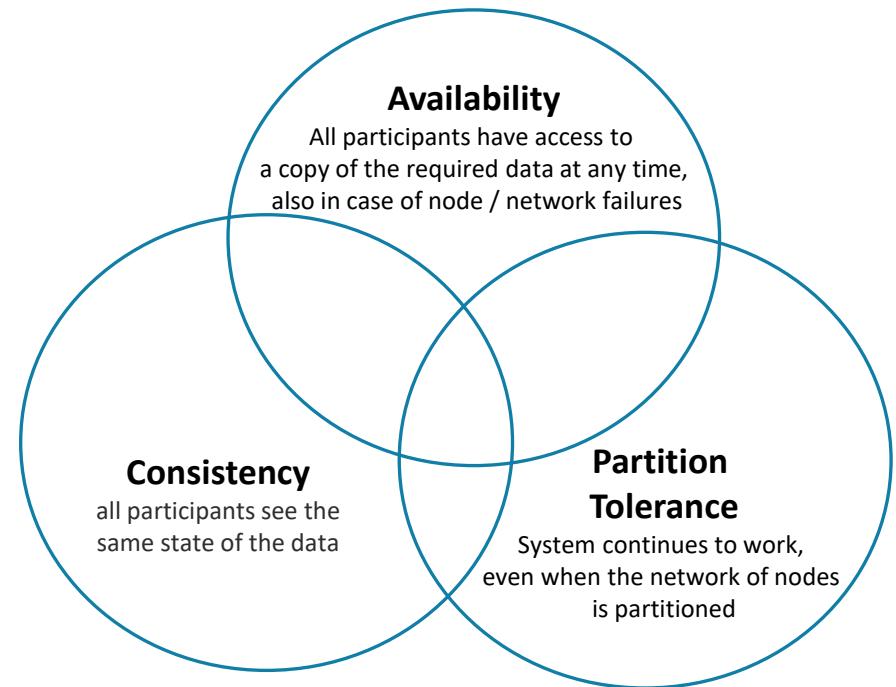
- Complex TX management with ACID properties not necessary for many applications
 - TXs are managed at application level
 - Optimistic concurrency control often more efficient
- Distributing a RDBMS is a hard job
 - For most RDBMS no out-of-the-box solutions for distributed/partitioned databases exist
 - Complex setup & maintenance of distributed RDBMS
 - Configuration changes often require downtime
→ not acceptable in online business!



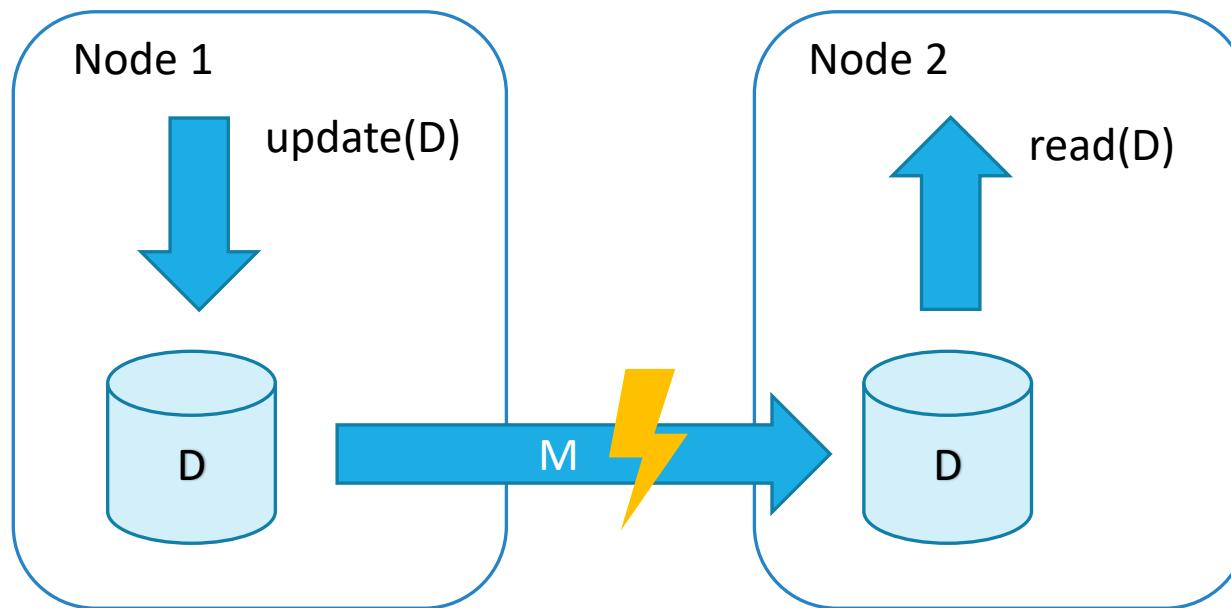
The CAP Theorem

- **Consistency, Availability, Partition Tolerance**
- In networked shared-data systems **can only guarantee two** [Brewer, 2000]

- During a network partition, choose between consistency and availability
 - RDBMS choose *consistency*
 - NoSQL databases choose *availability*



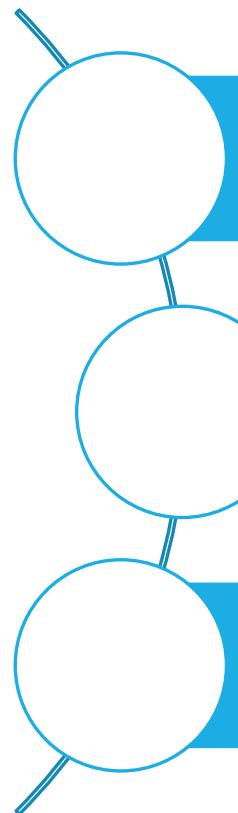
CAP – Example



[Rahm & Sattler, 2015]

Systems are not that simple anymore, there are many means to handle partitioning while maximizing availability and consistency suited for the application at hand [Brewer, 2012]

Consistency Models



Strong consistency

- Changes are immediately communicated to all nodes

Eventual Consistency

- Guarantees that nodes will eventually get the changes

Weak consistency

- Cannot guarantee that changes have been communicated



Quiz

<https://www.menti.com>
Code **4960 2893**



NoSQL Database Systems

Not
Only SQL

NoSQL: Does **NOT** provide an **SQL** interface, **Not Only SQL**

No relational
data model

Distributed
running well on
clusters, horizontal
scalability

Schemaless
→ more flexible

Built for web
applications

Working with
replications

BASE Principle

Basically Available

Availability more important than consistency

Soft State

Continuous changing states
→ eventual consistency due to availability

Eventual consistent

If no new updates are made, guarantees that the data will be consistent eventually

Data Models of NoSQL Systems

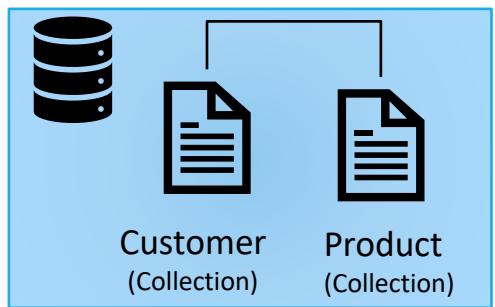
„One-size-fits-all“ no longer applicable!

Data model types

- **Document-oriented Data Model**
- Key-Value Data Model
- Column-oriented Data Model
- Graph Data Model

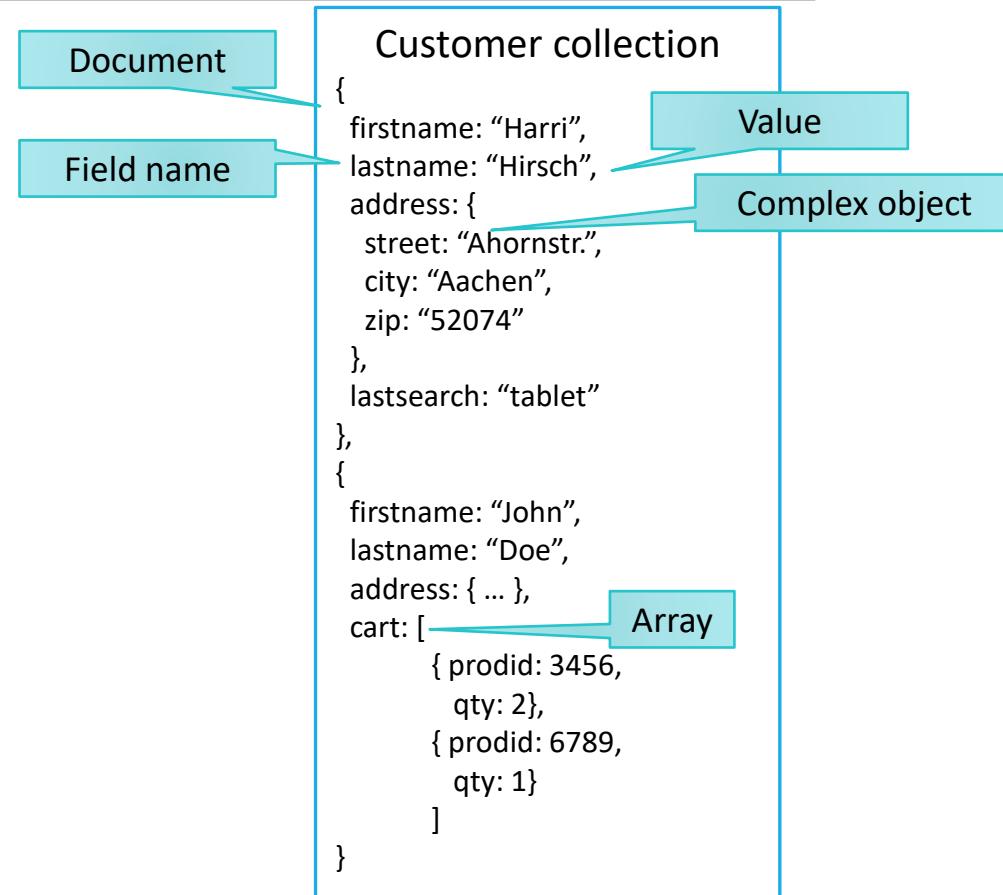
Advantages of NoSQL data models

- Complex objects not scattered across several relations
- Expensive join queries are avoided
- Easy mapping to application data model
- Increases programmers' productivity



Document-oriented Data Model

- **Document store (database):**
data stored in collections of similar documents
 - **Document:**
list of attribute-value pairs with possible nesting
 - Semi-structured self-describing data model → no schema definition necessary, documents in a collection may differ
 - Can be represented in XML, but mostly JSON is used
 - Indexes for efficient querying



Document-oriented Data Model

- Systems: **MongoDB**, CouchDB
- Good for:
 - Event logging
 - Content management & blogging systems
 - E-commerce applications
- Not well suited for:
 - Queries with varying result structure



Example: MongoDB

- Documents are stored in BSON (Binary JSON)
- Create database: db = client[dbname]
- Create collection:
db.createCollection("project", { capped : true, size : 1310720, max : 500 })
- ObjectId field _id: User- or system-generated
- Normalized or denormalized design
 - Normalized: use Ids to reference other objects / documents also in other collections
 - Denormalized: use aggregates to encapsulate other objects

{
 firstname: "John",
 lastname: "Doe",
 status:"Prime",
 address: { ... },
 cart: [
 {
 {
 prodid : "3456",
 prolname: "Fundamentals of Databases",
 pages: 1089,
 authors : [...]
 }
 qty: 2
 },
 {
 prodid : "1234",
 prolname: "Cow Book",
 pages: 889,
 authors : [...]
 }
 qty: 1
],
}

{
 firstname: "John",
 lastname: "Doe",
 address: { ... },
 status: "Prime",
 cart: [
 { "prodid": 3456,
 qty: 2},
 { prodid: 6789,
 qty: 1}
]
}

Queries in mongoDB - find

```
SELECT firstname, lastname  
FROM customer  
WHERE status = "Prime"
```

```
db.collection.find(<Pattern or conditions>, <result structure>)
```

```
result = db.customer.find({"status" : "Prime"},  
    {  
        "firstname" : 1,  
        "lastname" : 1  
    })
```

Returns a cursor
for documents

```
{  
    firstname: "John",  
    lastname: "Doe",  
    address: { ... },  
    status:"Prime",  
    cart: [  
        {  
            {  
                prodid : "3456",  
                prodname: "Fundamentals of Databases",  
                pages: 1089,  
                authors : [...]  
            }  
            qty: 2  
        },  
        {  
            prodid : "1234",  
            prodname: "Cow Book",  
            pages: 889,  
            authors : [...]  
        }  
        qty: 1  
    ],  
}
```

Queries in mongoDB - find

Condition with comparisons

```
result = db.customer.find( { 'cart.qty': {$gte:2} } )
```

Multiple conditions

```
result = db.customer.find( { $elemMatch: { 'cart.qty': {$gte:2, $lt:5} } } )
```

1: asc, -1: desc

Query modifiers: sorting

```
result = db.customer.find( { 'cart.qty': {$gte:2} } ).sort( {lastname:-1} )
```

Query modifiers: count

```
result = db.customer.find( { 'cart.qty': {$gte:2} } ).count()
```

```
{  
    firstname: "John",  
    lastname: "Doe",  
    address: { ... },  
    status: "Prime",  
    cart: [  
        {  
            {  
                prodid : "3456",  
                prodname: "Fundamentals of Databases",  
                pages: 1089,  
                authors : [...]  
            }  
            qty: 2  
        },  
        {  
            prodid : "1234",  
            prodname: "Cow Book",  
            pages: 889,  
            authors : [...]  
        }  
        qty: 1  
    ]  
}
```

Queries in mongoDB – Aggregation Pipelines

```
db.collection.aggregate[<stage1>, <stage2>, ...])
```

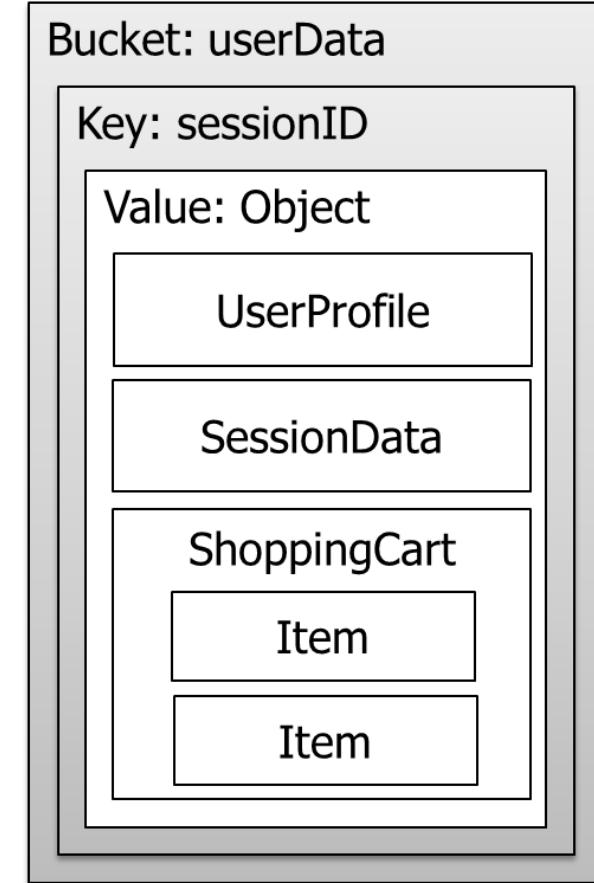
- Nested query stages (collection is input to stage 1, stage 1 is input to stage 2...)
- Each stage performs an operation on the input documents (filter, grouping, calculations)
- Pipeline can return results for groups (avg, sum, max, min, total)

```
SELECT COUNT(*)  
FROM customer  
WHERE country = „Germany“  
GROUP BY status
```

```
result = db.customer.aggregate(  
    [  
        {$match: { country : „Germany“ } },  
        {$group: { _id : „$status“}, count:{$sum: 1}}  
    ]  
)
```

Key-Value Data Model

- **Key:** identify data items uniquely and rapidly
- **Value:** depends on the system, can be simple byte strings, structured, unstructured, semi-structured data items
- Primary goal: Rapid access by keys
- Objects can be linked to other objects
- But: No standard query languages



Key-Value Data Model

- Systems: Riak, Redis, **DynamoDB**, Voldemort
- Good for
 - Session information
 - User profiles
 - Shopping carts
- Not well suited for:
 - Query by data



riak



redis



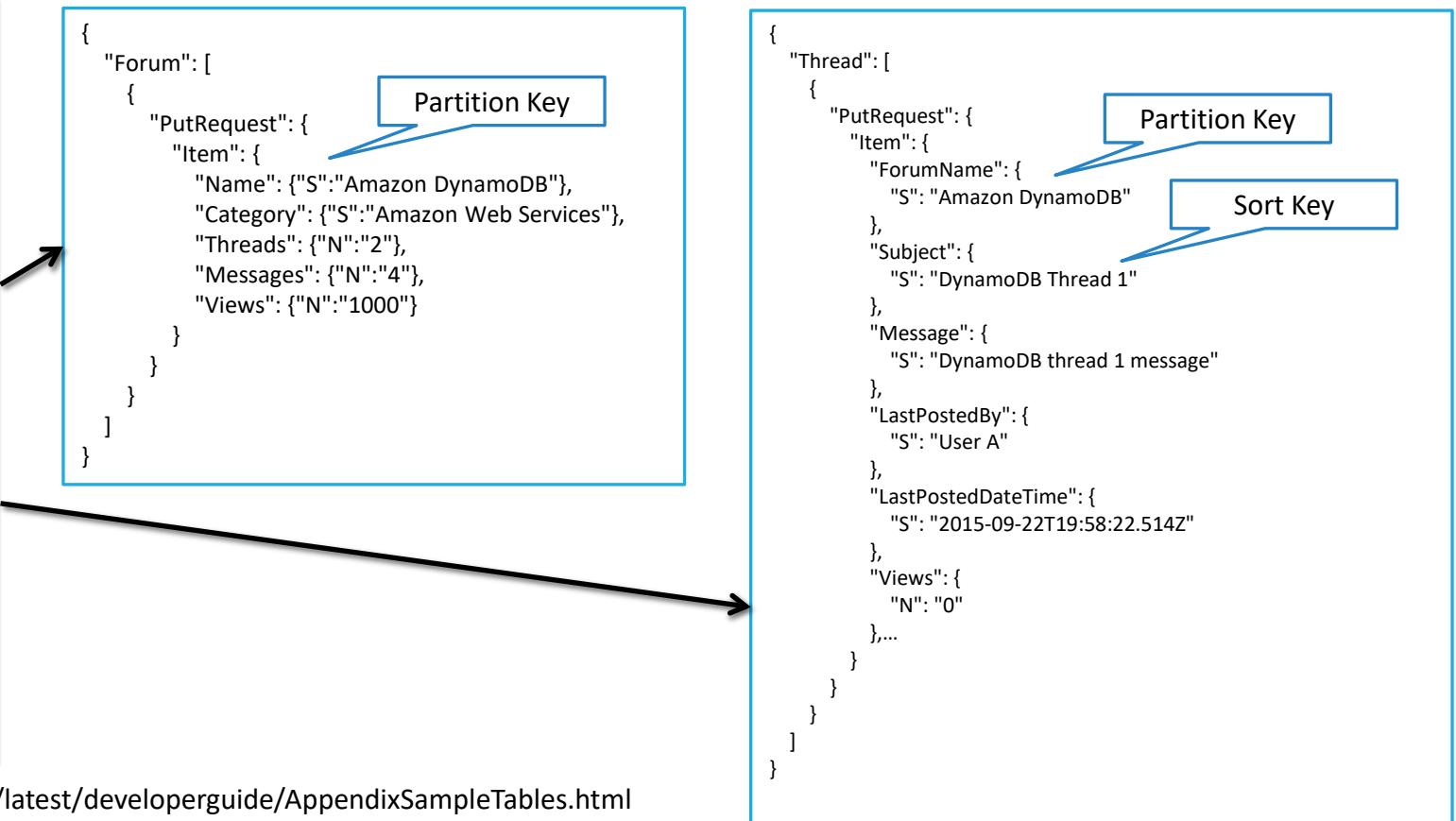


Example - DynamoDB

- Amazon product, available in AWS as storage component
- Data model is organized in tables, items, and attributes
- **Table:** collection of self-describing items (objects) with a primary key
- **Primary key**
 - Partition key: single attribute → unique, hash function determines partition for storage
 - Partition key and sort key (composite PK):
 - pair <A,B> of attributes; A is used for hashing, B orders items of A with same value for A
 - Example: versioning of items with <ItemID, timestamp>
- **Item**
 - Contains the actual values
 - number of (attribute, value) pairs, formulated in JSON (only protocol, not the internal storage format)

Example – DynamoDB Forum Application

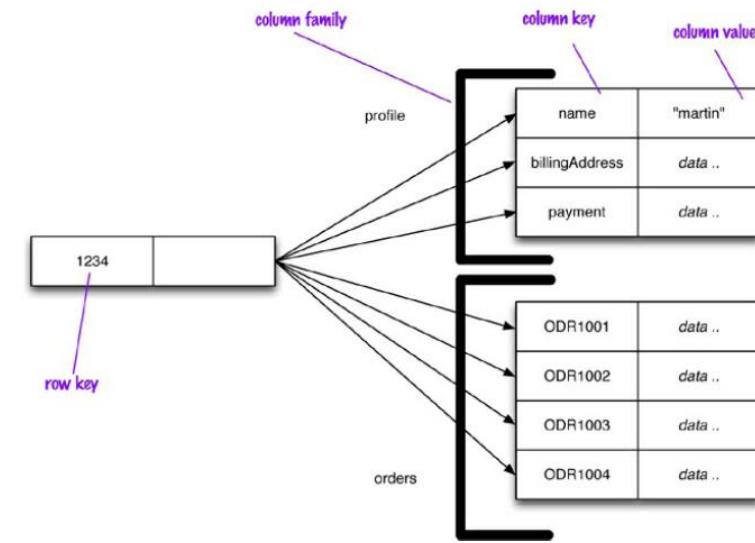
Table Name	Primary Key
<i>ProductCatalog</i>	Simple primary key: <ul style="list-style-type: none"><i>Id</i> (Number)
<i>Forum</i>	Simple primary key: <ul style="list-style-type: none"><i>Name</i> (String)
<i>Thread</i>	Composite primary key: <ul style="list-style-type: none"><i>ForumName</i> (String)<i>Subject</i> (String)
<i>Reply</i>	Composite primary key: <ul style="list-style-type: none"><i>Id</i> (String)<i>ReplyDateTime</i> (String)



<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AppendixSampleTables.html>

Column-based / Wide Column Data Model

- Similar to key-value data model, but multidimensional key
- **Key:** usually combination of table name, row key, column, and timestamp
- **Column:** composed of column family & column key
- **Row:** one map for multiple column families → sparse multidimensional, distributed, persistent sorted map
- **Map:** collection of (key,value) pairs, i.e., key is „mapped“ to the value
- Not just a relational data model with a column-oriented storage
- Systems: Google BigTable, Apache HBase, Cassandra



Relational	Column-oriented DB
Database	Namespace
Table	Column Family
Row	Row
Column (same for all rows)	Column (can be different per row)

Example: HBase

<https://hbase.apache.org/book.html#datamodel>

- **Namespace:** Collection of tables
- **Table:** Consists of multiple rows
- **Row**
 - Consists of row key and one or more columns with values
 - Sorted alphabetically by row key → related rows should be close to each other (see domain example)
- **Column**
 - Combination of (column family:column qualifier)
 - Qualifier → used for indexing, mutable

Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8	contents:html = "<html>..."		
"com.cnn.www"	t5	contents:html = "<html>..."		
"com.example.www"	t5	contents:html = "<html>..."		people:author = "John Doe"

Example: HBase

<https://hbase.apache.org/book.html#datamodel>

- **Column Family**

- physical collocation of columns and values
- Each column family is stored in its own files
- Fixed at creation of namespace

- Multiple versions of data items stored using timestamps

- **Cell:** holds basic data item identified by key

- **Keys:** combination of table, rowid, column family, column qualifier, and timestamp

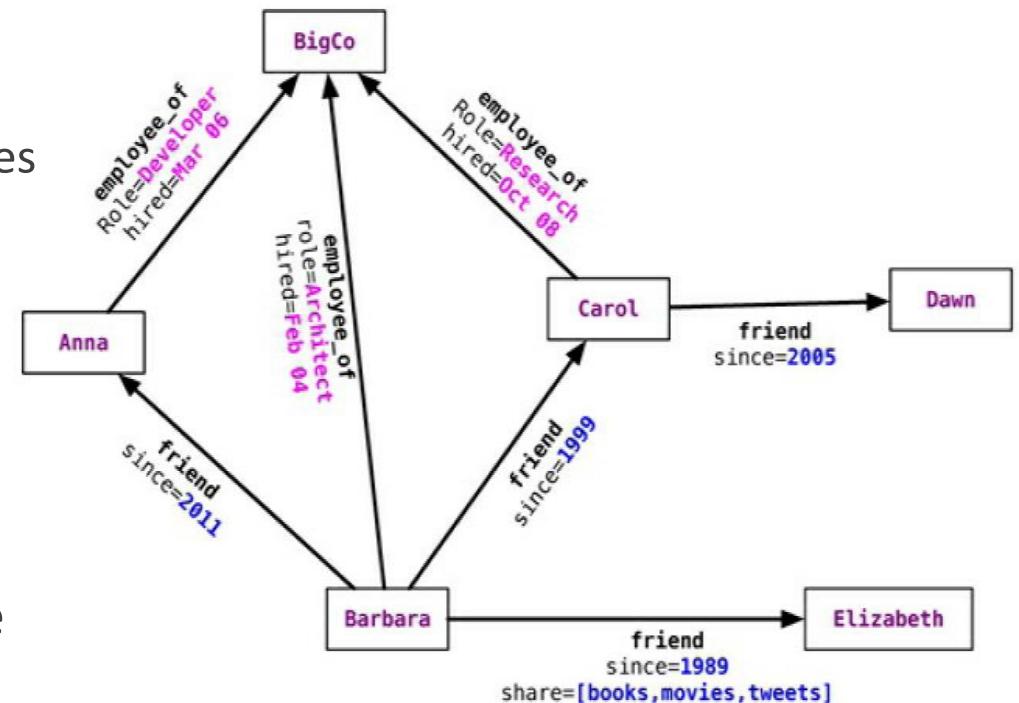
Conceptual table view				
Row Key	Time Stamp	ColumnFamily contents	ColumnFamily anchor	ColumnFamily people
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"	
"com.cnn.www"	t8	contents:html = "<html>..."		
"com.cnn.www"	t5	contents:html = "<html>..."		
"com.example.www"	t5	contents:html = "<html>..."		people:author = "John Doe"

- Physical storage view

Row Key	Time Stamp	Column Family
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

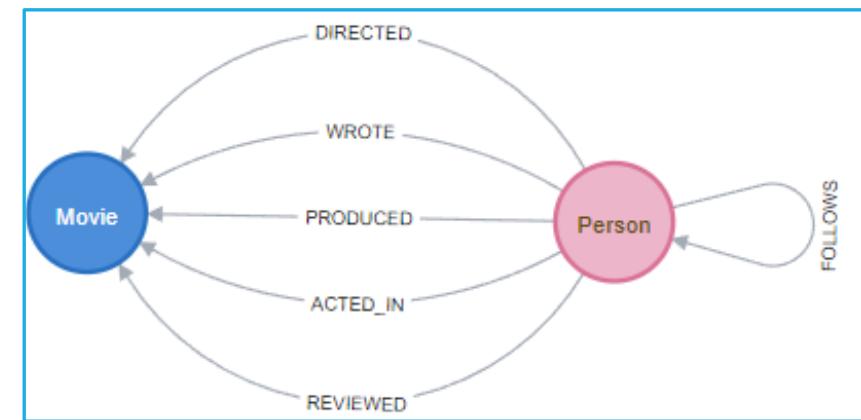
Graph Data Model

- Data represented as graphs
 - Collection of nodes and edges
 - Nodes and edges can have properties → indicate what entities they represent
- Can store data with both, individual nodes & edges
- Edges have directions
- Queries traverse graph structure using path expressions
- Stronger consistency and transaction model than in other NoSQL systems
- Systems: **Neo4j**, OrientDB, Amazon Neptune



Example: Neo4j

- Data items: stored as nodes and relationships with properties
- Nodes
 - Can have no, one, or multiple labels
 - Nodes with the same label are grouped into a collection
→ useful for querying
- Relationships
 - Directed → have start and end node
 - Have a type → indicate similar relationships for querying
- Properties
 - Specified via a map pattern composed of one or more (name, value) pairs, e.g.,



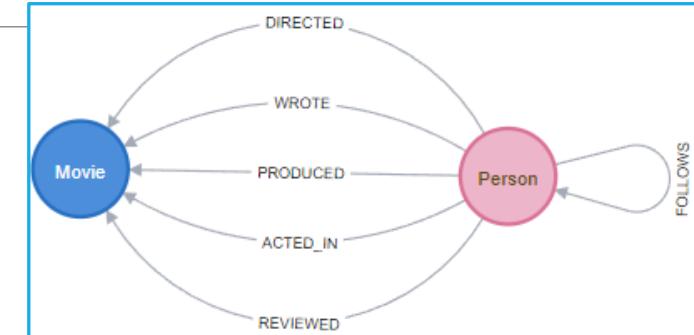
<https://neo4j.com/developer/example-project/>

Example: Neo4j

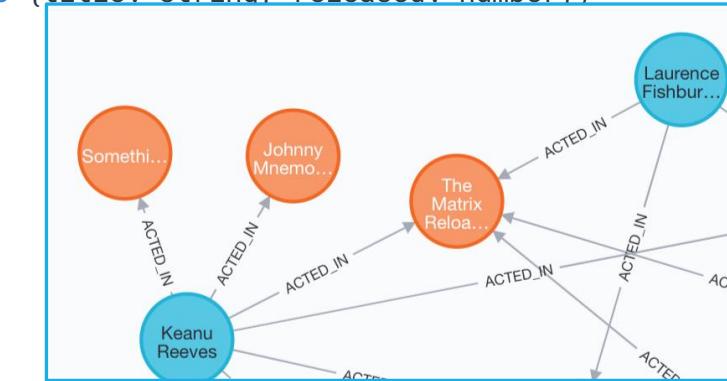
- **Path**

- Traversal of a part of the graph → typically part of a query
- Specifies a pattern → data that matches the pattern will be retrieved
- Start node, one or multiple relationships, one or more end nodes
- Optionally, a schema can be defined
→ indexes and constraints based on labels and properties
- Cypher Query Language: declarative, pattern-based language with possibility to query directed and undirected relationships

<https://neo4j.com/developer/example-project/>



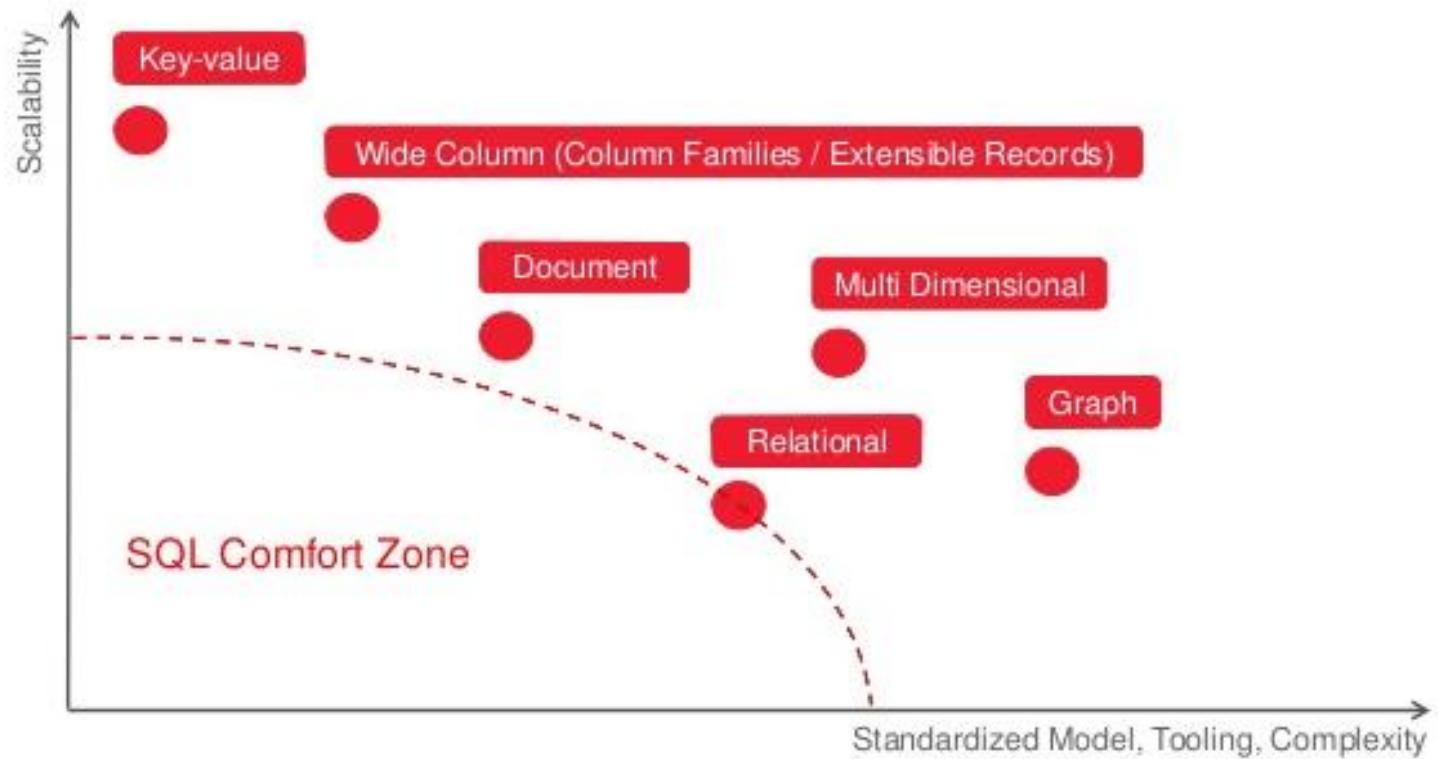
```
CREATE (:Person {name: string})-[:ACTED_IN {roles: [string]}]-> (:Movie {title: string, released: number})
```



```
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(movie:Movie) RETURN movie
```

Classification of Data Models

[https://www.slideshare.net/gschmutz/
sql-vs-nosql-43786447](https://www.slideshare.net/gschmutz/sql-vs-nosql-43786447)





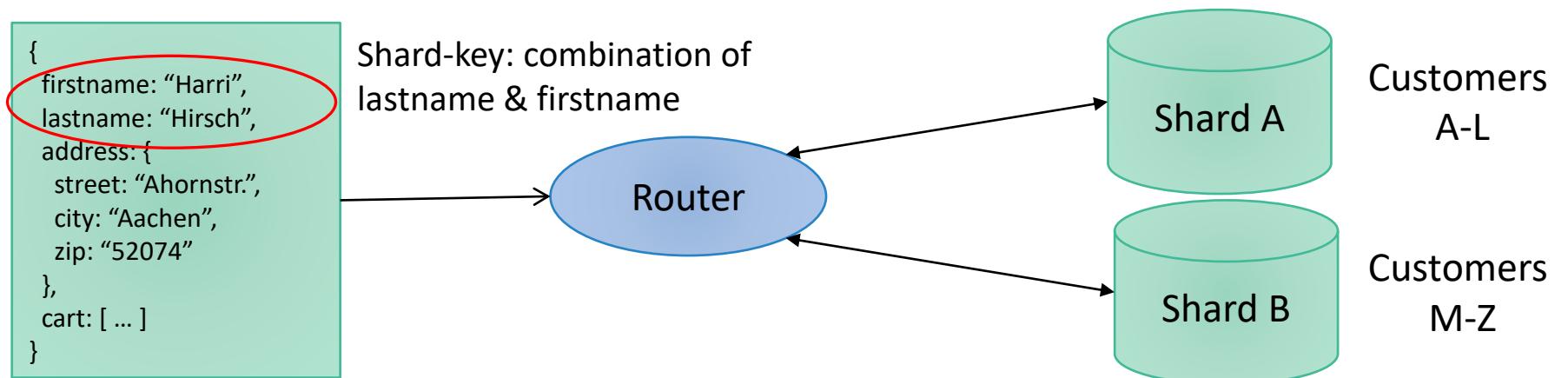
Quiz

<https://www.menti.com>
Code **4960 2893**



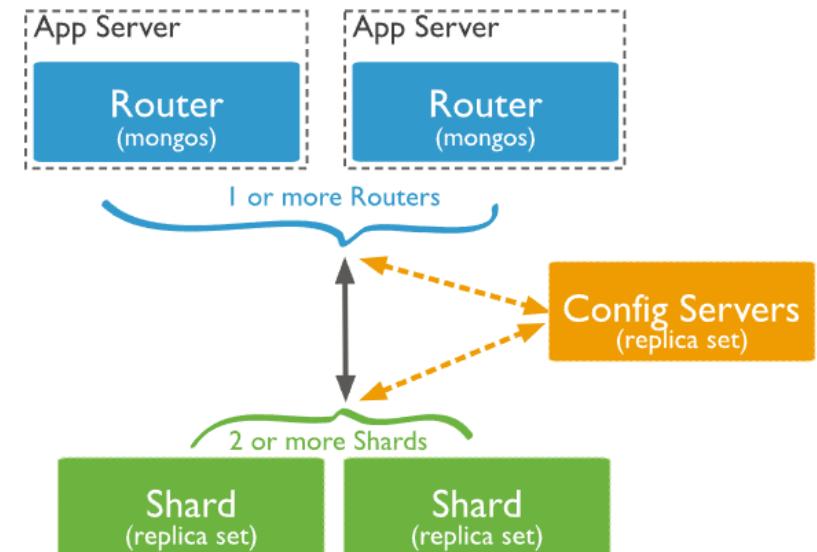
Sharding

- **Horizontal partitioning:** disjoint subsets of data are distributed to different servers
→ increases performance of reads and writes, load balancing
- Related data should be stored in one server
- Data should be evenly distributed between servers
- NoSQL systems support *auto-sharding*, which requires definition of a *shard-key*
- Sharding can be combined with replication



Example: Sharding in MongoDB

- **Shards:** documents are divided into disjoint chunks
- Stores shards on different nodes for load balancing
- Each node only processes operations for documents in own shard
- **Query router:** knows which nodes contain which shards and routes query to that nodes



<https://docs.mongodb.com/manual/sharding/>

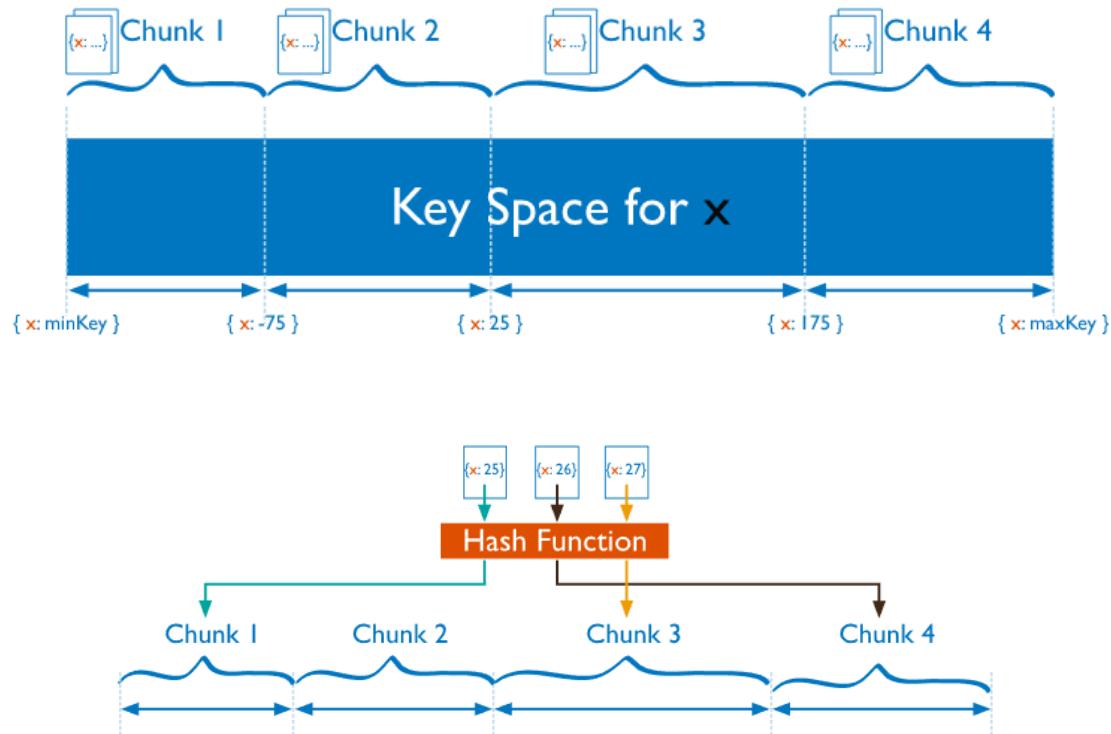
Example: Sharding in MongoDB

- **Shard key**

- Document field for partitioning, specified by user
- Must exist in every document and have an index
- Shard key values are divided into chunks by partitioning scheme

- **Partitioning schemes**

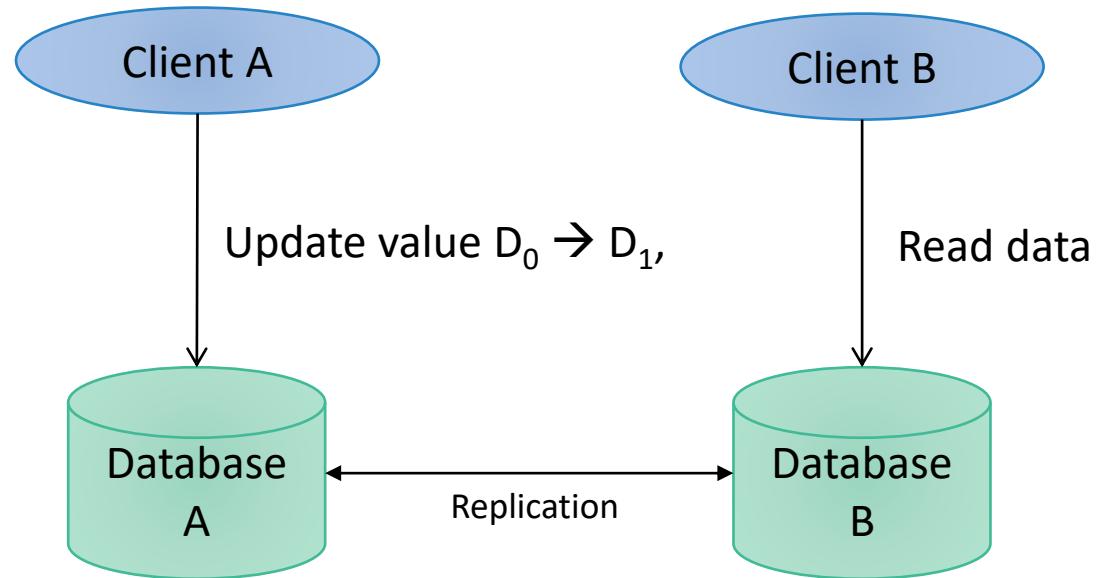
- Range partitioning: divides values into equal ranges
→ better for range queries
- Hash partitioning: applies hash function to each key
→ better for equality selections



<https://docs.mongodb.com/manual/sharding/>

Eventual Consistency

- Replication reduces performance as data is written to multiple nodes
→ Inefficient as other TXs have to wait until data is committed in all nodes
- **Eventual consistency:** inconsistent intermediate data possible, as changes are replicated with delay
- Variants [Rahm & Sattler, 2015]
 - **Causal consistency:** only reads possible, when a message about update has send from A to B
 - **Read-your-writes:** guarantees that all own updates can be read by client A
 - **Session consistency:** read-your-writes in one session
 - **Monotonic reads:** if process reads D_k it will never read older versions of it
 - **Monotonic writes:** serialization of write operations is guaranteed



Schemaless



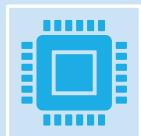
Most NoSQL systems do not require a schema definition, but an application based on NoSQL databases is not really schemaless



Schema is defined at a different level

Application code → Implicitly defined in the code, e.g., checking for attributes

Web services → Explicitly defined by WSDL / XML Schema



Advantage of NoSQL systems are their flexibility and ease-of-use with respect to schemas

However, schema changes can be done in SQL, too

Summary

- Classical RDBMS did not meet the requirements of web applications, especially in terms of scalability and fault tolerance
- NoSQL systems offer simple, but yet powerful mechanisms for scalability and programmability
- However, in terms of abstraction and declarative query languages, they are a step backwards
- Be careful when looking at performance charts, they are sometimes comparing apples & oranges
- Most systems are best for at least one certain application

Polyglot persistence: Organizations will use more than one DBMS technology!



4.2 Big Data Systems & Architectures

Learning Goals

At the end of this section you will be able to

- ✓ know and explain the 6 V's of big data
- ✓ explain and setup Hadoop and know its components
- ✓ explain the HDFS characteristics, its architecture and components
- ✓ know and explain the Lambda architecture and its layers
- ✓ explain and apply the map-reduce pattern in an example



Big Data - 6 V's or more...

Validity, Vulnerability, Volatility,
Visualization ...

Volume

Amount of data from many sources

Variety

types of data, such as structured, semi-structured, unstructured

Velocity

The speed at which the data is generated

Veracity

The quality of the data

Value

The business value of the data

Variability

the ways in which the data can be used and formatted

Big Data Applications



Marketing

What is interesting for our customers?

What is the reaction of the customer to certain marketing actions?



Production („Industrie 4.0“)

Quality Control

Process Optimization

Predictive Maintenance



Logistics

Optimization of storage and product streams



Controlling/Finance

Prediction

Risk Analysis



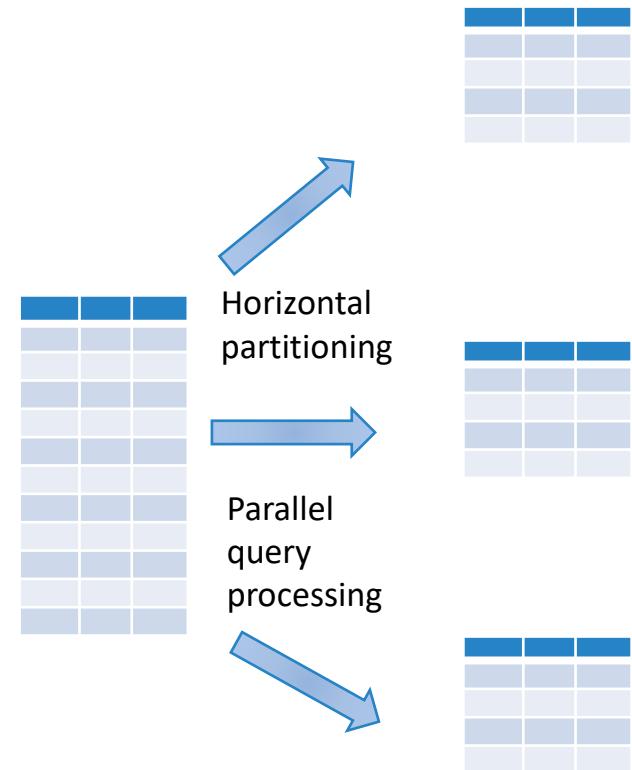
IT

Performance Analysis

Security

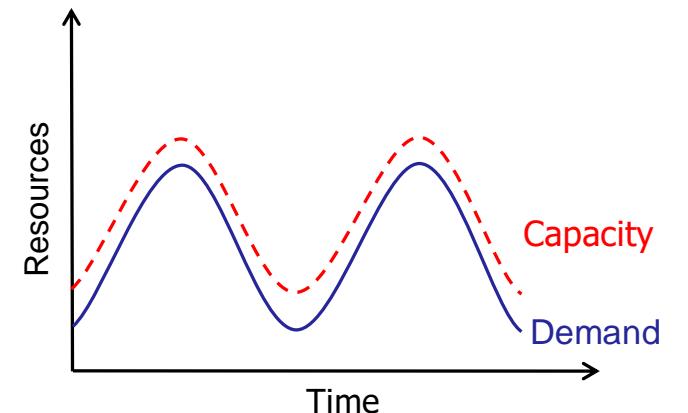
Historical Perspective: (Big) Data Processing before ~2005

- Relational DBMS are dominating the DBMS market
- Parallel DBMS for large scale data processing
 - Shared-nothing architecture
 - Horizontal partitioning
 - Massive parallel processing technology (MPP)
 - Well-designed schema and distribution
 - Declarative query processing with SQL
 - Transaction management based on ACID
 - Parallelization & distribution is transparent for user
 - Example systems: Teradata, IBM Netezza, Greenplum, ...
(costs >>1m EUR)



Limitations of (Parallel) Relational DBMS

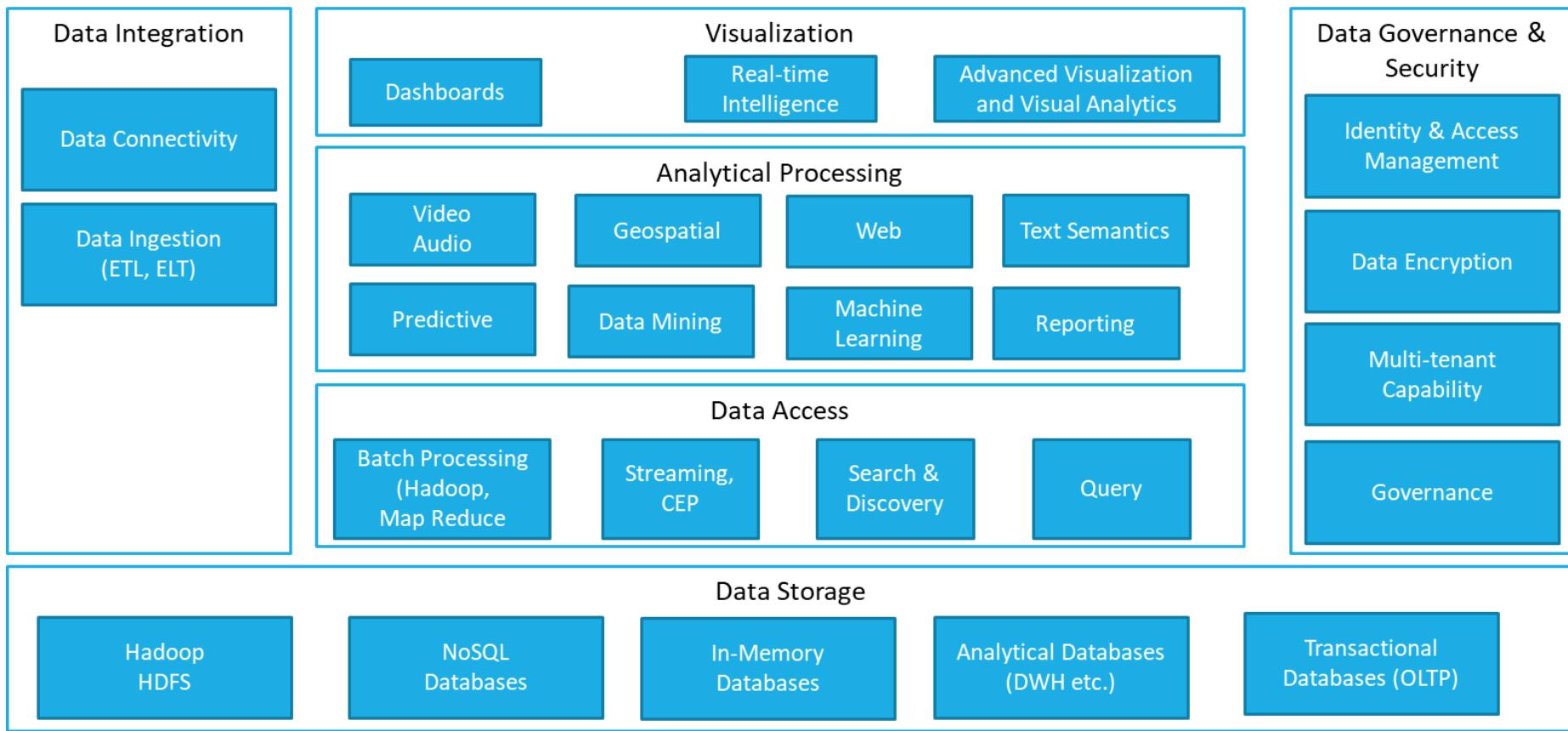
- Parallel Relational DBMS are good when
 - Applications are mostly known a-priori (schema is somehow fixed)
 - You need complex queries across relations, expressible in SQL
- In the area of Big Data, you need systems with support for
 - Flexible schemas and semi-structured data
 - Complex data analytics
 - Ad-hoc analytics with short setup times
 - Easy scalability
 - Fault tolerance



Big Data Architectures

- Big Data requires large, scalable, distributed architectures
- Heterogeneity
 - Sources
 - Systems
 - Requirements
 - Client Applications
- Complex eco-systems with several independent components to be integrated
- Hadoop
 - Is not a Big Data system, it is just one (but important) component
 - “[..] is a **framework** that allows for the **distributed processing** of **large data sets** across clusters of computers using **simple programming models.**[..]”¹
 - Provides the basic underlying infrastructure for many Big Data systems

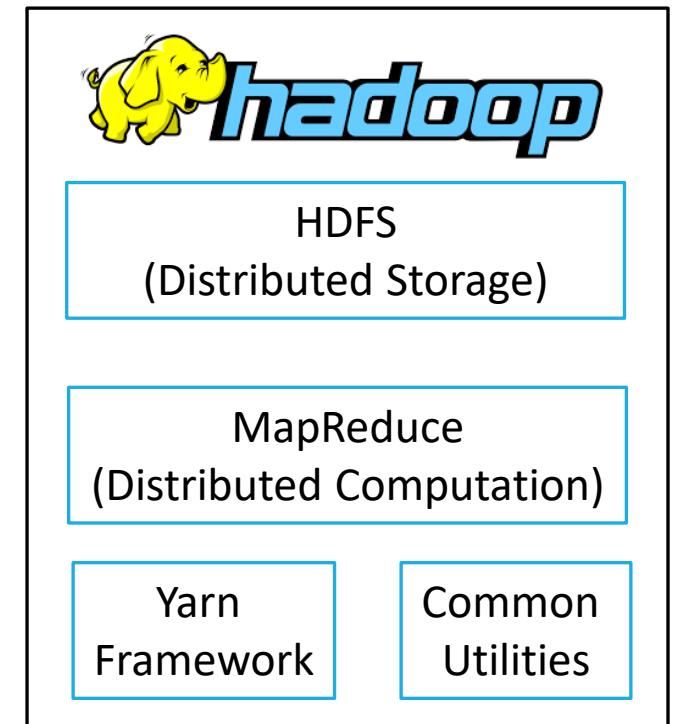
Big Data Technologies



[Bitkom, 2014]

4.2.1 Hadoop - The first Big Data System

- Inspired by Google's work on GFS (Google File System) and MapReduce (internally used by Google for data management)
- Hadoop 0.1 was released 2006, 1.0 in 2012, currently 3.3.4
- Major components
 - **HDFS** (Hadoop Distributed File System) as distributed data storage
 - **MapReduce** for efficient distributed computation
 - **YARN** (Yet another resource negotiator) for resource management
- Available as basic service in many cloud computing platforms (e.g., Azure, AWS, Google, ...)



https://www.tutorialspoint.com/hadoop/hadoop_introduction.htm

Hadoop: Some Success Stories

ebay

- 532 nodes cluster (8 * 532 cores, 5.3PB).
- Heavy usage of Java MapReduce, Apache Pig, Apache Hive, Apache HBase
- Using it for search optimization and research



Facebook

- Store copies of internal log and dimension data sources for reporting/analytics and ML
- Currently 2 major clusters:
 - A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
 - A 300-machine cluster with 2400 cores and about 3 PB raw storage.
 - Each (commodity) node has 8 cores and 12 TB of storage.
 - Built a higher-level data warehousing framework using Hive



<https://cwiki.apache.org/confluence/display/hadoop2/PoweredBy>

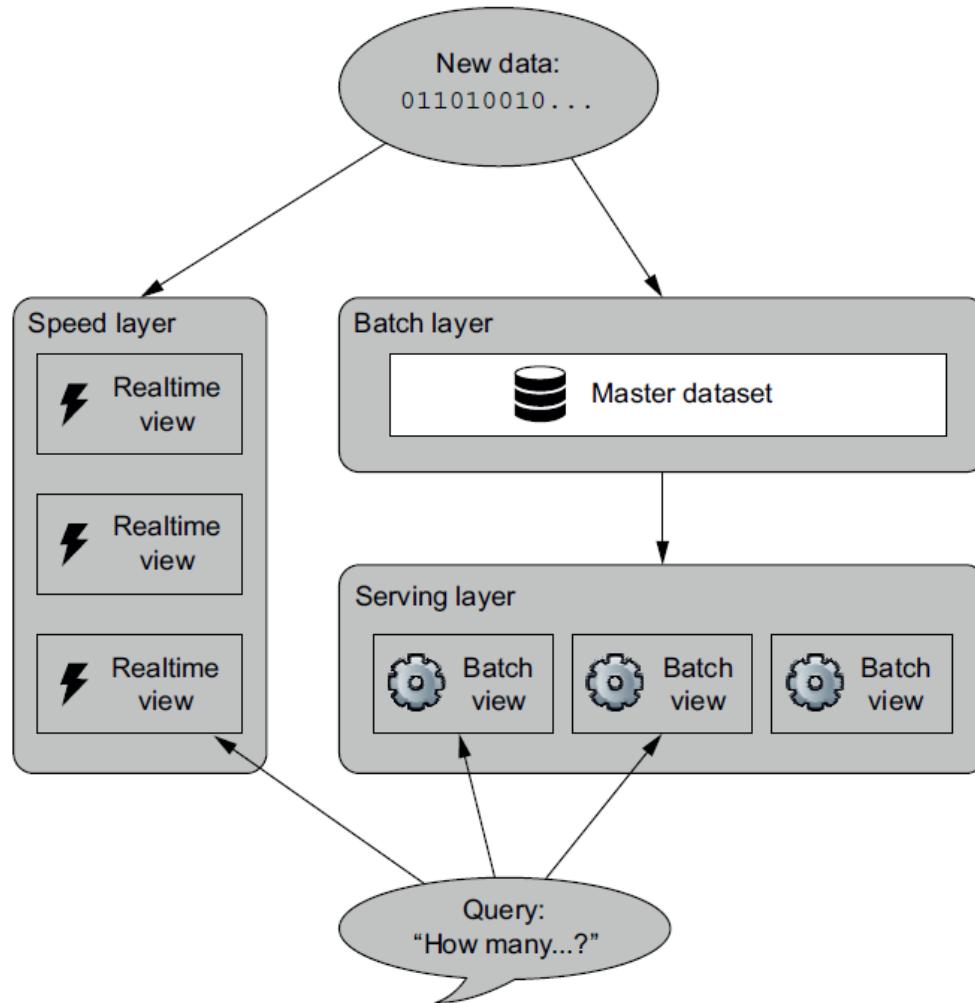
Data Processing in RDBMS vs. Hadoop

- **Relational DBMS (*Schema on write*)**

1. Design schema & create corresponding tables in DBMS
2. Load data: create mappings from sources to DB schema, define ETL (Extract, Transform, Load) processes to load the data into the database and execute them
 - Queries are possible, system is ready for users

- **Big Data Systems (*Schema on read*)**

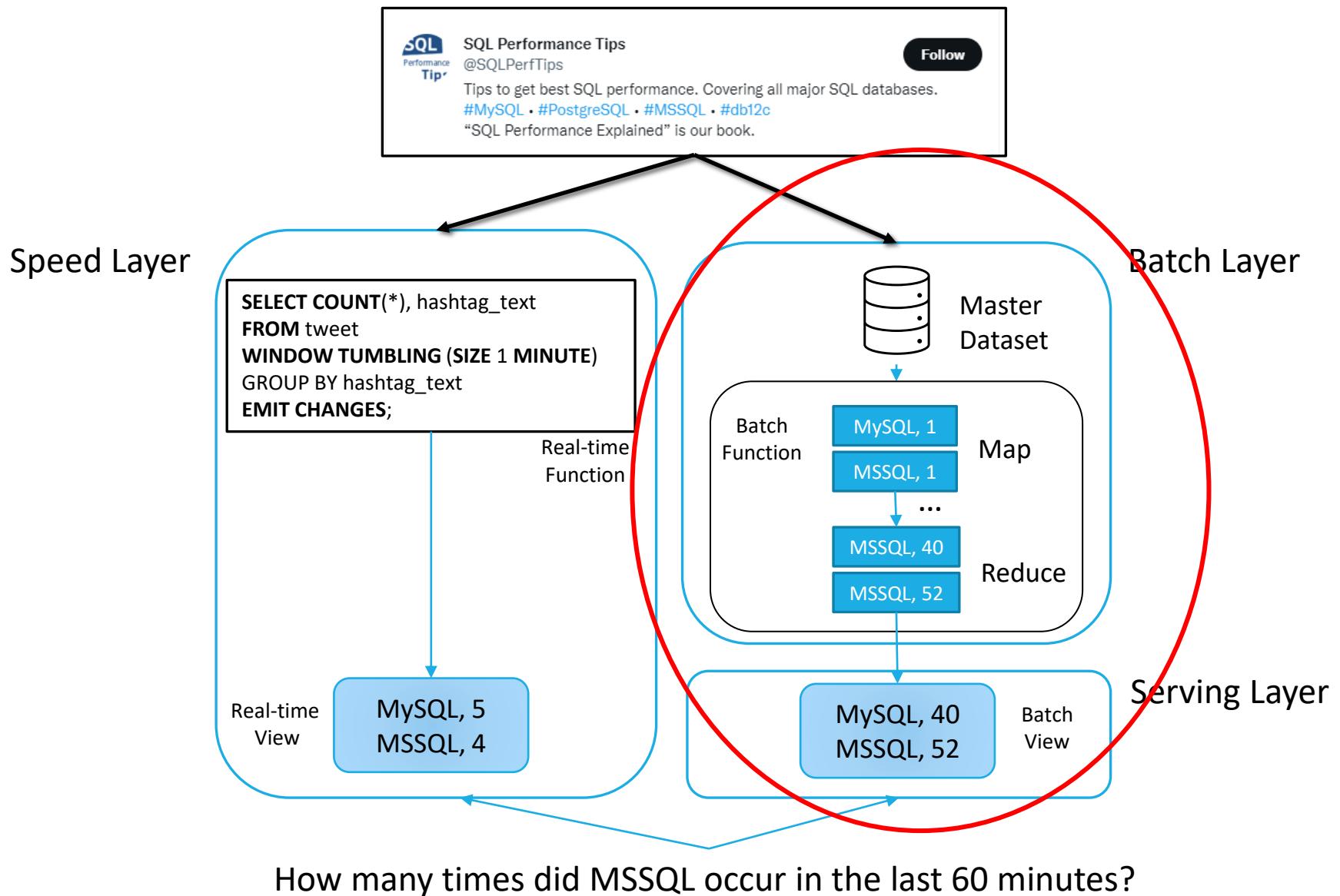
1. Load data: **Copy data as-is** into HDFS
2. Map SQL queries to data files
 - Queries are possible, system is ready for users
 - Shorter time until running system, although mapping of SQL queries to Map/Reduce jobs in Hadoop might be complex



Lambda Architecture

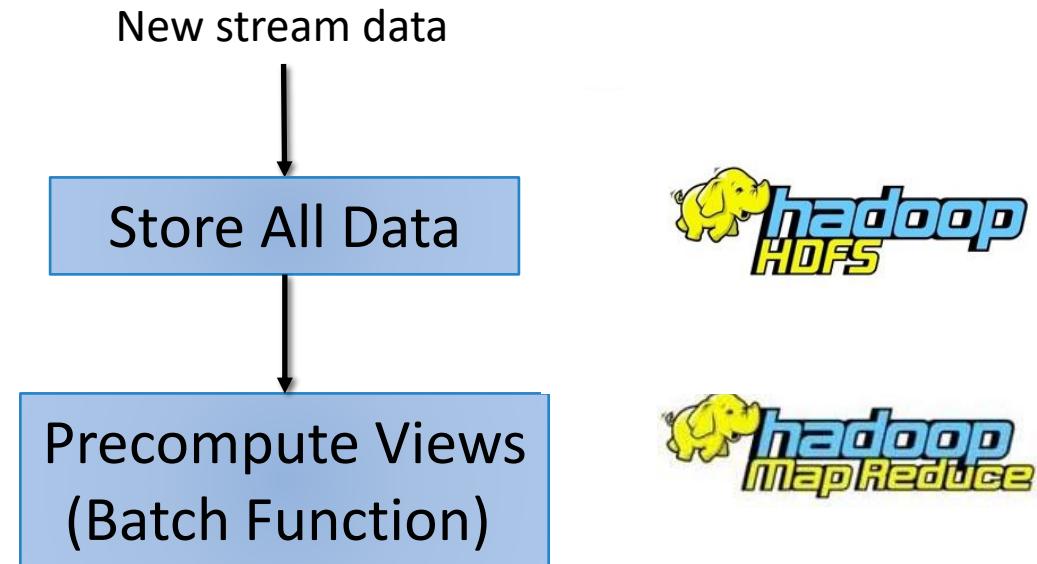
- **Goal:** All-time availability of *all* processed data while keeping up performance
- Can process massive amounts of data with a hybrid approach (batch & stream processing methods)
- Incoming data is dispatched to Batch Layer and Speed Layer
- Functions on both layers prepare views on serving layer for querying
- Users can query merged results

[Marz & Warren, 2015]



Lambda Architecture – Batch Layer

- Manages master data set: immutable, append-only set of raw data
- Distributed precomputation of batch views
- Processes **all** data



HDFS – Main Features

Block-oriented file storage distributed over many machines in a cluster

File sizes of GBs to TBs, tens of millions of files in a single instance

Immutable files: write-once-read-many, only appends and truncates possible

Highly **fault-tolerant** - recover quickly & automatically

Replication: by default, each data block is replicated three times

Batch-oriented processing with emphasis on high throughput

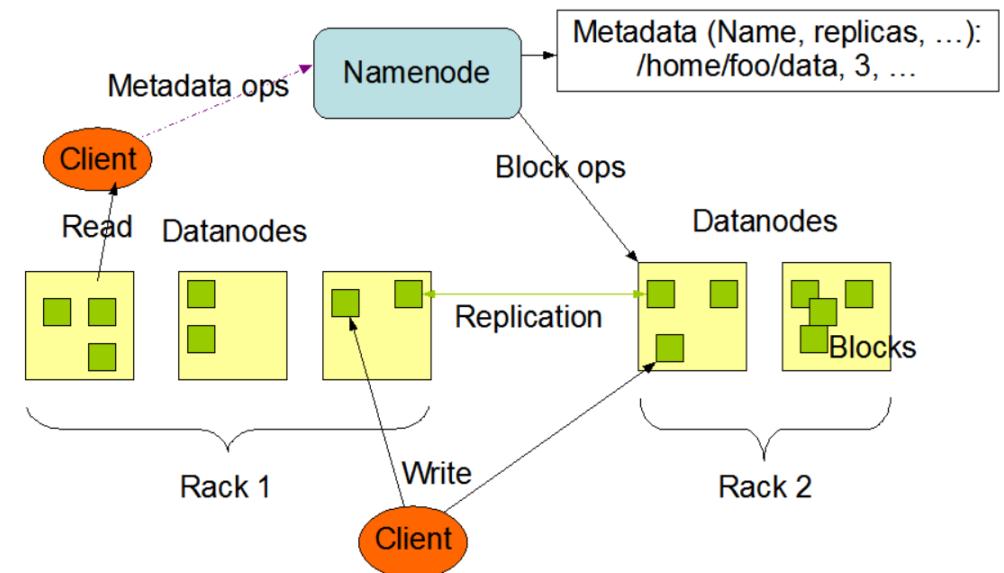
Designed to be deployed on commodity hardware

Computation is done „close to the data“ (→ MapReduce)

Portability: implemented in Java to be available on multiple platforms

HDFS Architecture - NameNode

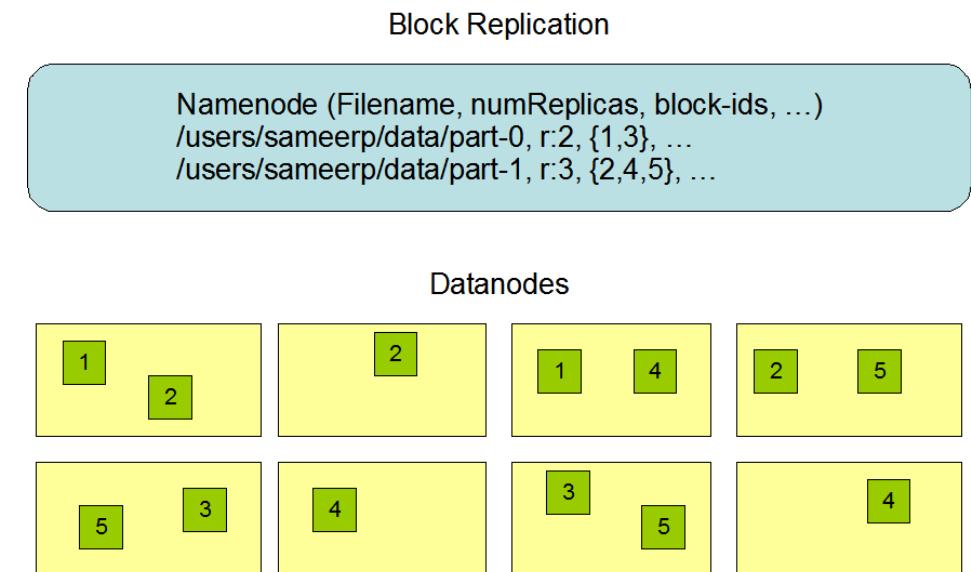
- Master-worker architecture
- Block-oriented storage of arbitrary files in hierarchical directory structure
- 1 NameNode, multiple DataNodes
- **Rack:** physical collection of nodes
- **NameNode**
 - Manages metadata (file information, blocks, replicas, ...)
 - Regulates access to files and directories by clients (open, close, rename)
 - Responsible for distribution & replication
 - Might be bottleneck in case of many small files
 - Multiple instances with automatic failover
- Hierarchical organization (directories, files,) in the file system namespace



<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>

HDFS Architecture - DataNodes

- File is divided into blocks
- Data block creation, deletion, replication as instructed by NameNode
- Serve read and write requests
- Send heartbeat to NameNode to detect failed nodes
- Block size (typically 128 MB) and replication factor are configurable per file
- Arbitrary file formats, but new formats (Parquet, AVRO, ORC, ...) are supported → e.g., for compression



<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>

Setup Pseudo-Distributed Single Node Cluster

- Download binary from <http://hadoop.apache.org/>
- Follow instructions on
<http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html>
- Format filesystem to HDFS:

```
$ bin/hdfs namenode -format
```
- Start the nodes: \$ sbin/start-dfs.sh
- Browse the NameNode at <http://localhost:9870/>
- Run MapReduce jobs packed in jars

Hadoop Overview Datanodes Datanode Volume Failures Snapshot Startup Progress Utilities ▾

Overview 'localhost:9000' (✓ active)

Started:	Fri Dec 02 15:55:33 +0100 2022
Version:	3.3.4, ra585a73c3e02ac62350c136643a5e7f6095a3dbb
Compiled:	Fri Jul 29 14:32:00 +0200 2022 by stevel from branch-3.3.4
Cluster ID:	CID-71ed176a-edea-434b-a41b-da2865484b8a
Block Pool ID:	BP-626747149-127.0.1.1-1669992075109

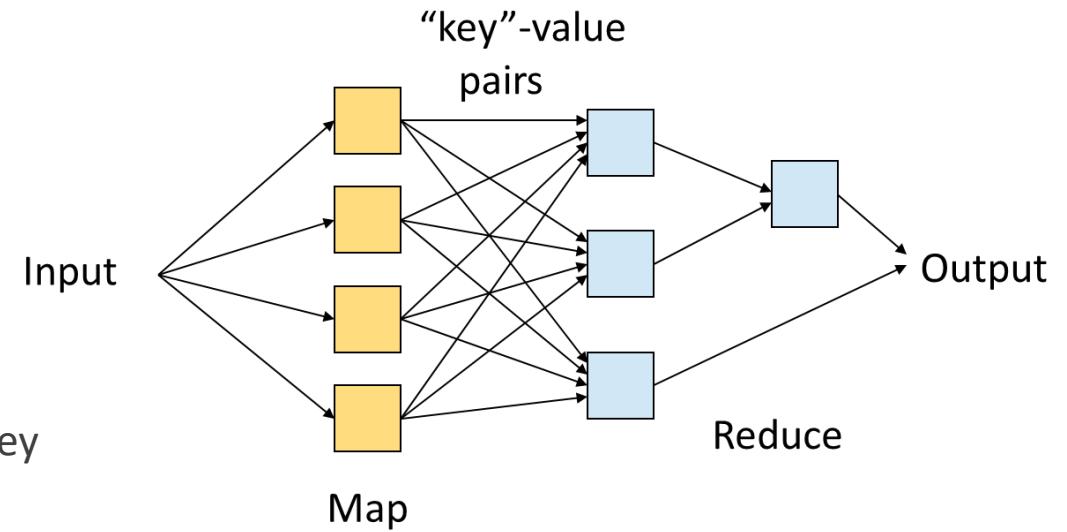
Summary

Security is off.
Safemode is off.
18 files and directories, 11 blocks (11 replicated blocks, 0 erasure coded block groups) = 29 total filesystem object(s).
Heap Memory used 184.71 MB of 734 MB Heap Memory. Max Heap Memory is 7.81 GB.
Non Heap Memory used 82.07 MB of 85.31 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

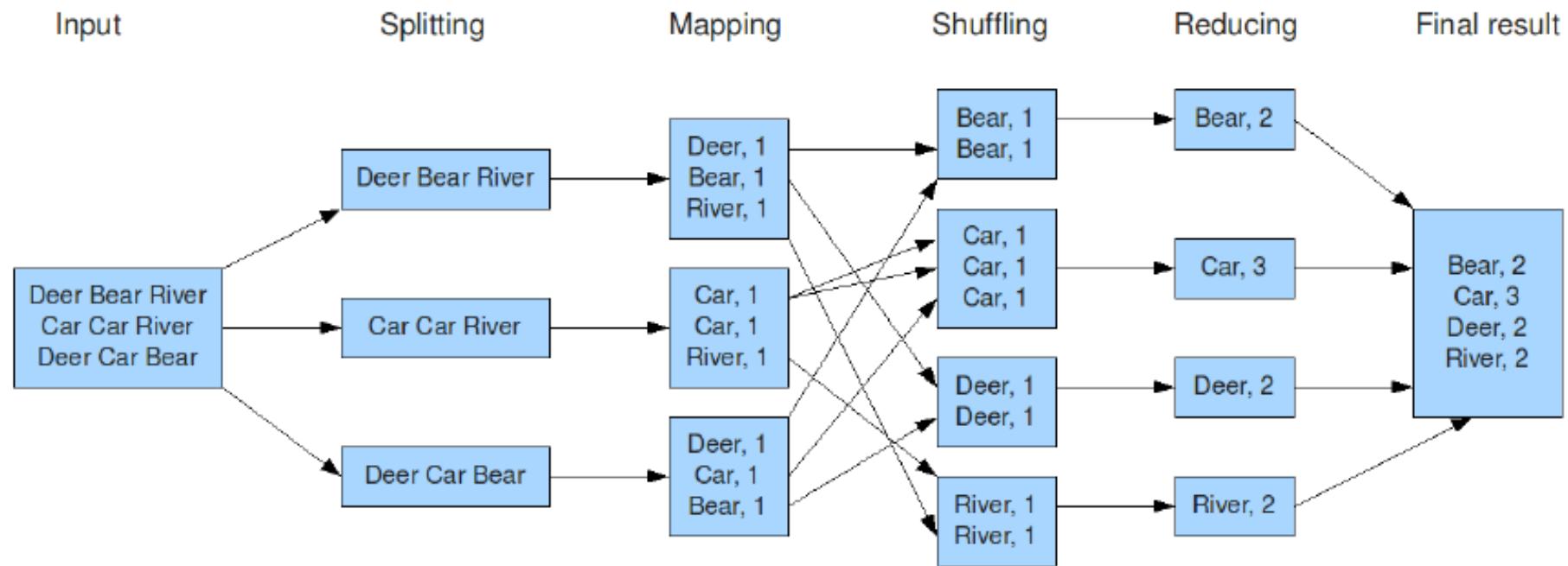
Configured Capacity:	952.62 GB
Configured Remote Capacity:	0 B
DFS Used:	56 KB (0%)
Non DFS Used:	289.58 GB
DFS Remaining:	663.05 GB (69.6%)
Block Pool Used:	56 KB (0%)

MapReduce

- Programming pattern for parallel computation in distributed system
- Inspired by primitives from functional programming
→ idea of higher-order functions
- **Function Map (data) → (key, value)**
 - Reads input data and emits key-value pairs
 - Keys are not necessarily unique in emitted pairs
- **Function Reduce (key, values) → (key, values)**
 - Gets input from Map function = set of values for a single key
 - Input and output structure should be the same
- Map and Reduce jobs can run on different nodes
- Application fields: big unstructured data (e.g., texts) or difficulty with definition of a relational schema



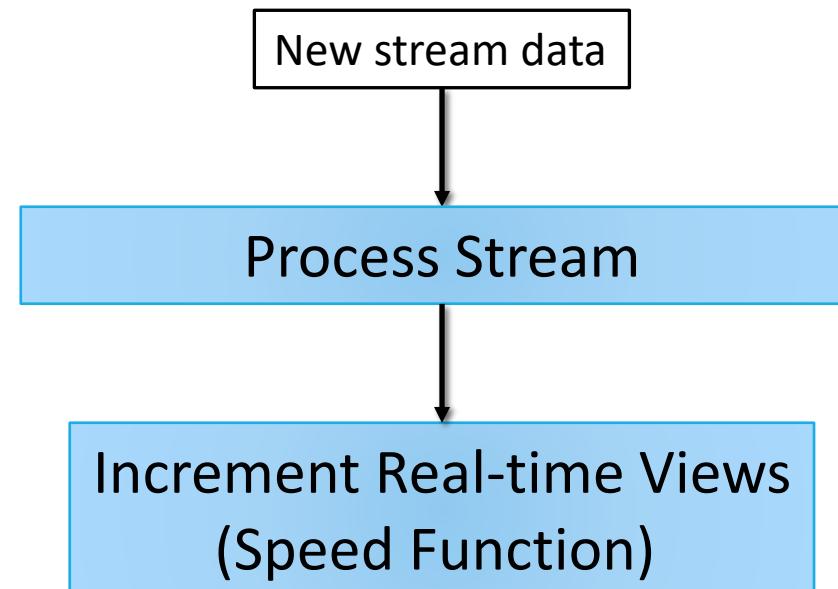
MapReduce Example: Word Count



Picture adapted from: Hadoop Mapreduce Framework in Big Data Analytics, Vidyullatha Pellakuri, Rajeswara Rao

Lambda Architecture – Speed Layer

- Computes **recent** data only
- Stores and updates real-time views
- Compensates high latency of updates at serving layer
- Not 100% accurate or complete



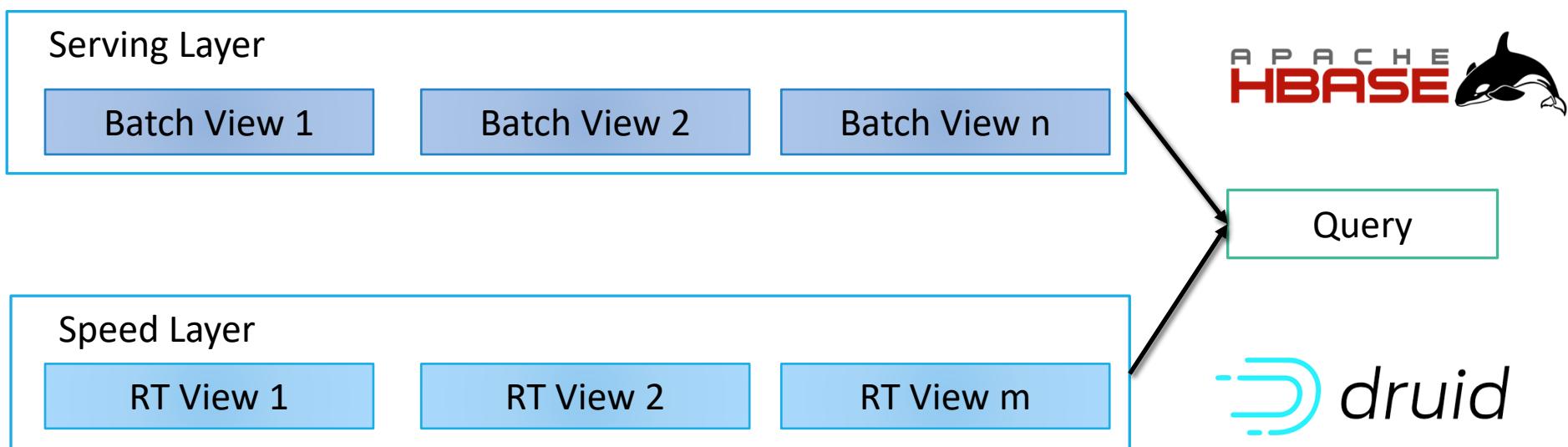
Apache Kafka

STORM

Spark Streaming

Lambda Architecture – Serving Layer

- Maintains & indexes batch views
- Combines batch and real-time views
- Provides ad-hoc query access to all data





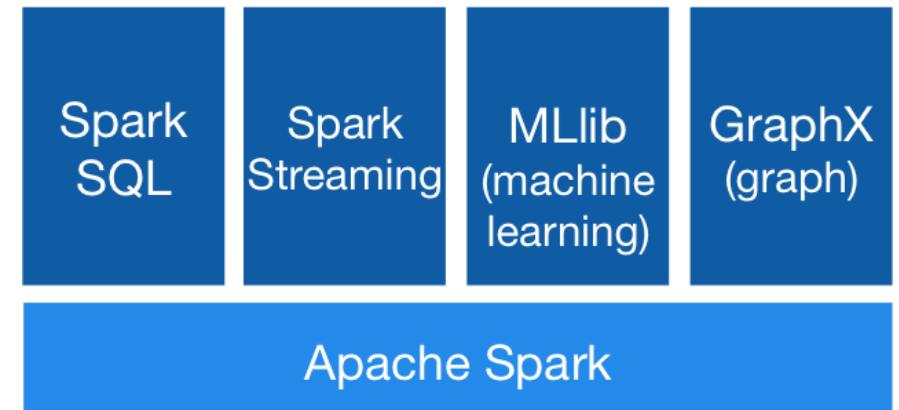
Quiz

<https://www.menti.com>
Code **4960 2893**



4.2.2 Apache Spark

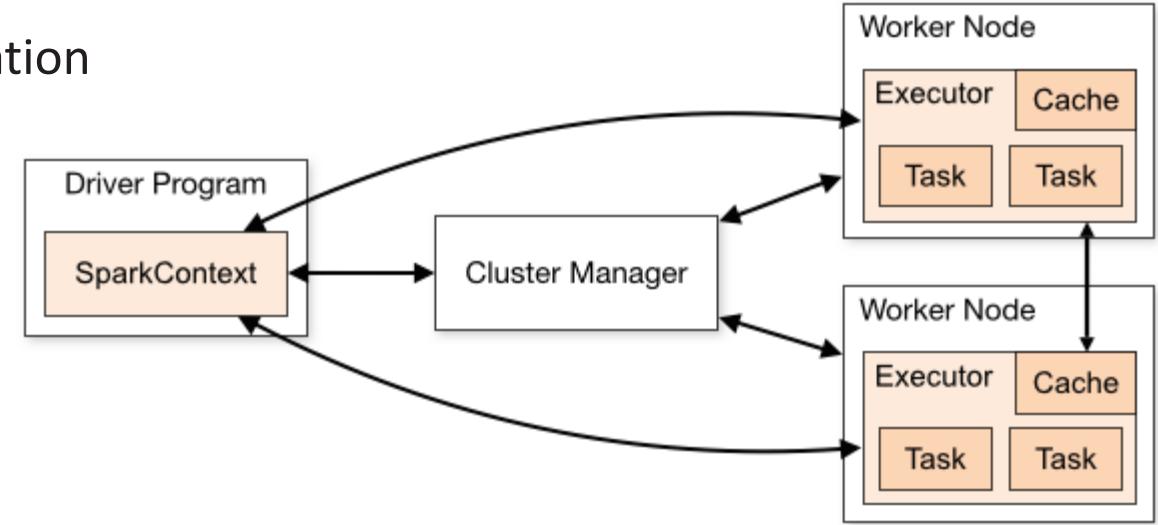
- Next-generation batch processing framework
- Uses a distributed data structure called RDD (Resilient Distributed Dataset)
- Full in-memory cluster computing
- Handles batch, interactive, and real-time processing within a single framework
- Native integration with Java, Python, Scala
- Combinable with Hadoop components
- More general: MapReduce just one supported construct
- Lazy evaluation



Spark Architecture

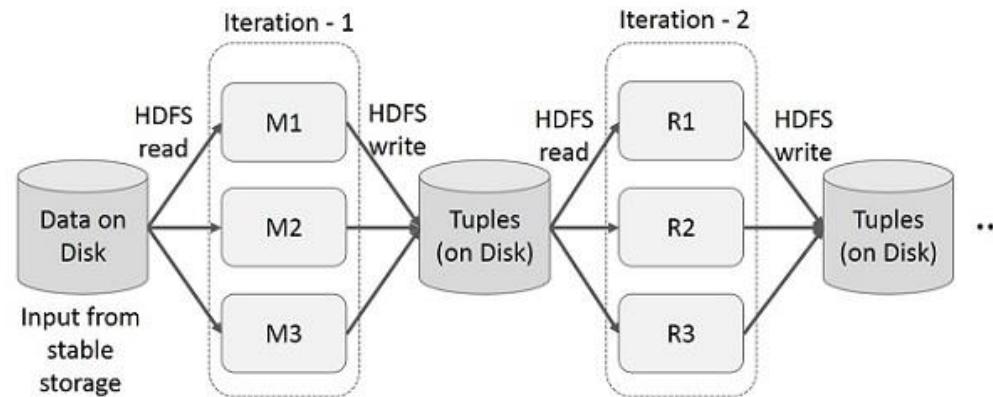
- Master-worker architecture
- **Driver**: central coordinator, has all information about the executors at any time
- **Worker Node**: consists of one or more Executor(s)
- **Executor**
 - responsible for running the task
 - register themselves at the Driver
- **Spark Application**: combination of drivers and workers
- **Cluster Manager**: launches Spark Application and manages the resources

<https://blog.knoldus.com/understanding-the-working-of-spark-driver-and-executor/>



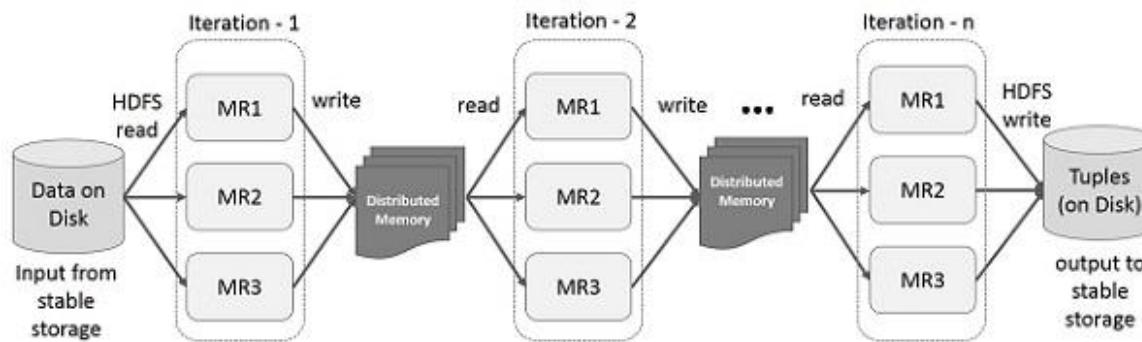
Data Processing in Hadoop vs. Spark

Hadoop



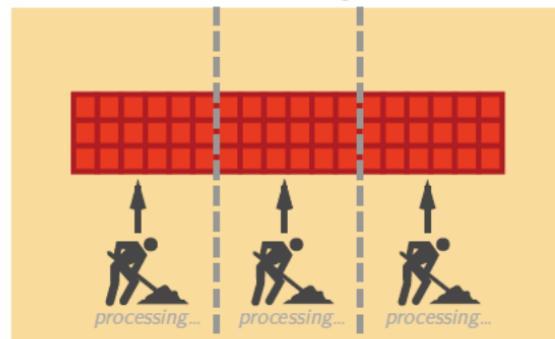
MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms

Spark

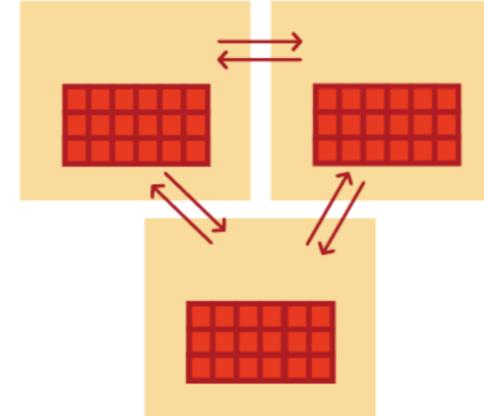


Data-Parallel vs. Distributed Data-Parallel

Shared memory:



Distributed:



- **Shared memory case:** Data-parallel programming model. Data partitioned in memory and operated upon in parallel
- **Distributed case:** Data-parallel programming model. Data partitioned between machines, network in between, operated upon in parallel

<http://heather.miller.am/teaching/cs212/slides/>

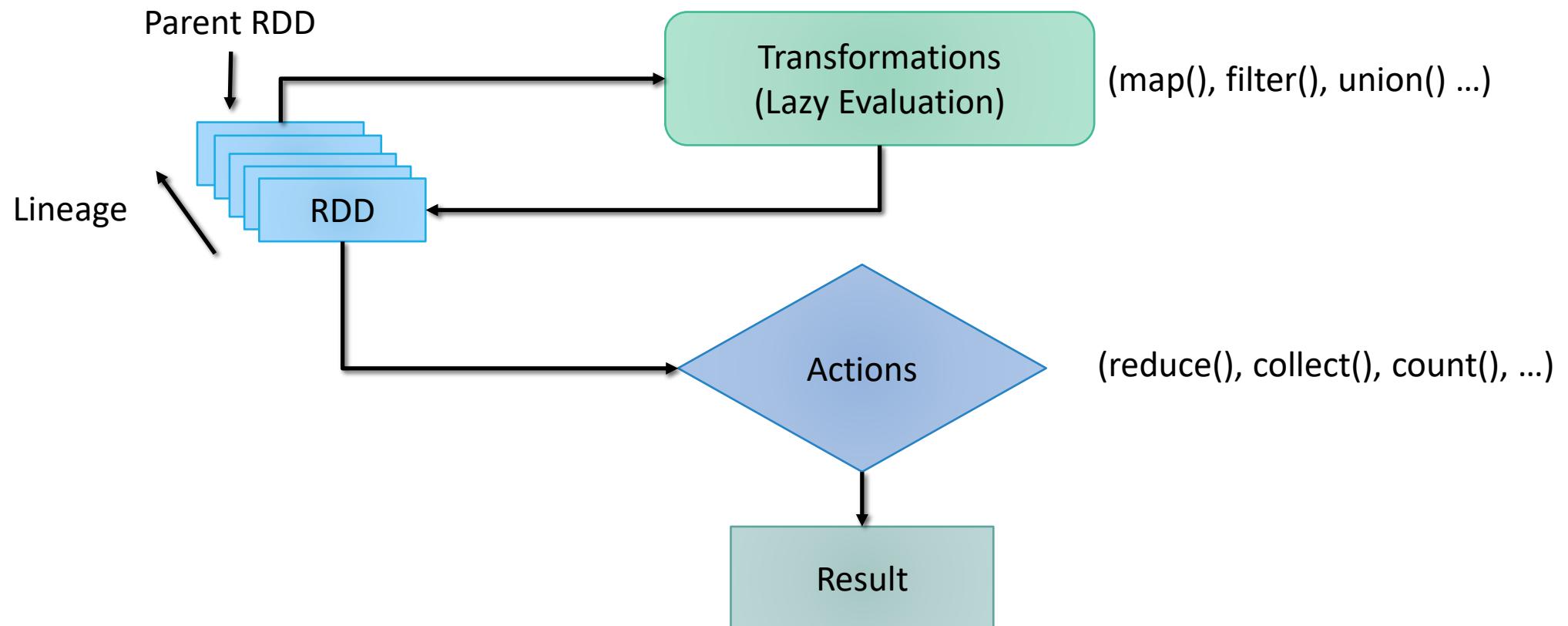
Resilient Distributed Datasets (RDD)

- Primary abstraction in Spark
- Fault-tolerant immutable, partitioned collection of objects (records)
 - Implementation of the distributed data-parallel model
 - Similar to (parallel) collections in Scala
- An RDD is a collection of data objects of any type (→ heterogeneity)
- There are also DataFrames (collection of row objects) and DataSets collection of typed-objects

Word Count in Apache Spark

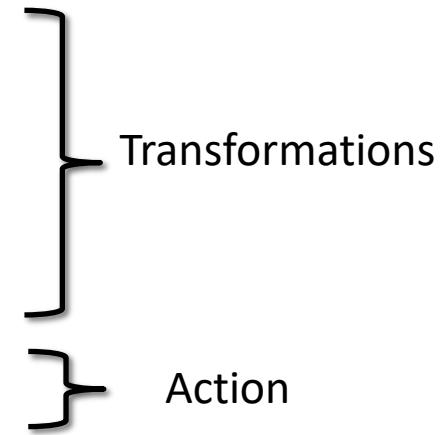
```
val rdd = spark.textFile("hdfs://...")  
  
val count = rdd.flatMap(line => line.split(" ")) //separate line into words  
    .map(word => (word, 1)) //include something to count  
    .reduceByKey(_ + _) //sum up the 1s in the pairs
```

RDD Operations



RDD Operations - Example

```
val largeList: List[String] = ...  
  
val wordsRdd = sc.parallelize(largeList)  
  
val lengthsRdd = wordsRdd.map(_.length)  
  
val totalChars = lengthsRdd.reduce(_ + _)
```



Persistence

- RDDs can be persisted in main memory (RAM) or disk
- Spark can persist (or cache) a dataset in memory across operations
- Each node stores in memory any slices computed, can be reused in other actions on that dataset
- Use of RAM or disk depends on available RAM and options of the transformation

Level	Space used	CPU time	In memory	On disk
MEMORY_ONLY	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK*	High	Medium	Some	Some
MEMORY_AND_DISK_SER [†]	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

Default

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

Partitioning

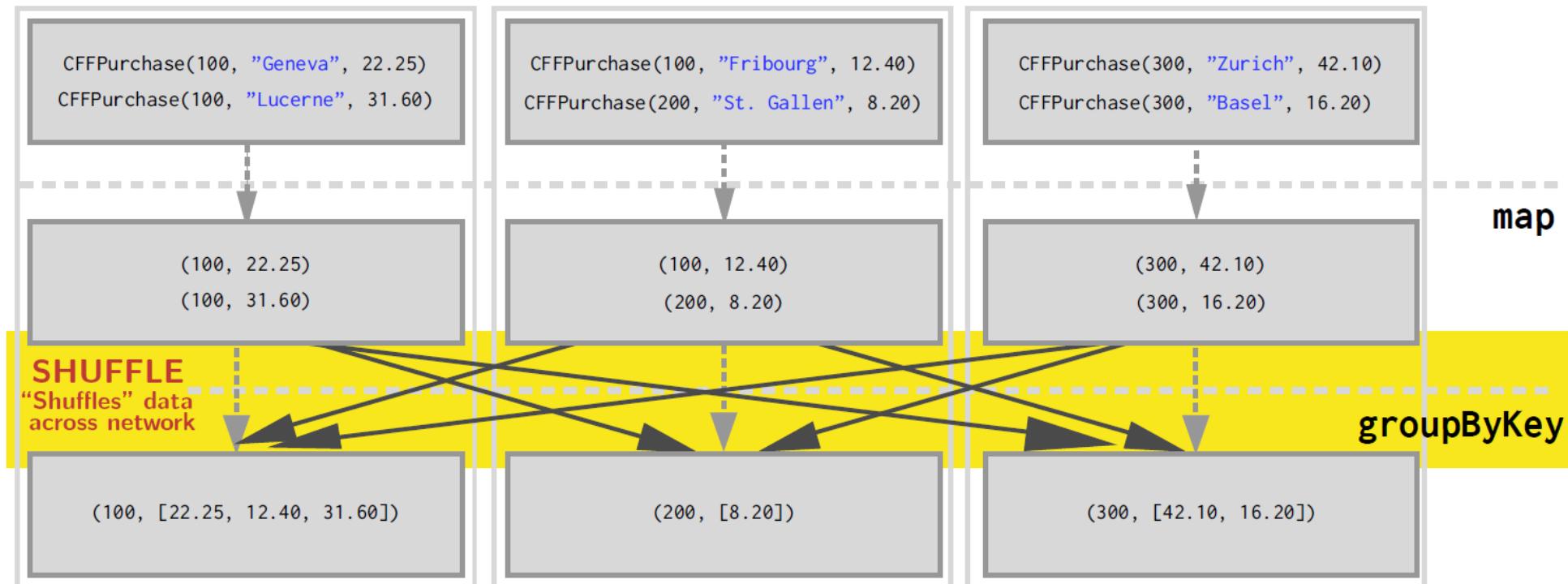
- The data within an RDD is split into several *partitions*.
- Properties of partitions:
 - Partitions never span multiple machines, i.e., tuples in the same partition are guaranteed to be on the same machine.
 - Each machine in the cluster contains one or more partitions.
 - The number of partitions to use is configurable. By default, it equals the *total number of cores on all executor nodes*.
- Partitioning methods
 - Hash partitioning
 - Range partitioning

Shuffling

- Shuffling moves RDDs between **different nodes** of a cluster
- It can occur for operations that cannot be processed within a single partition, e.g.,
 - `groupByKey`
 - `join`
 - `distinct`
 - `intersection`
 - ...
- As data has to be transferred between nodes, **network latency** is an issue
→ Shuffle operation should be avoided!

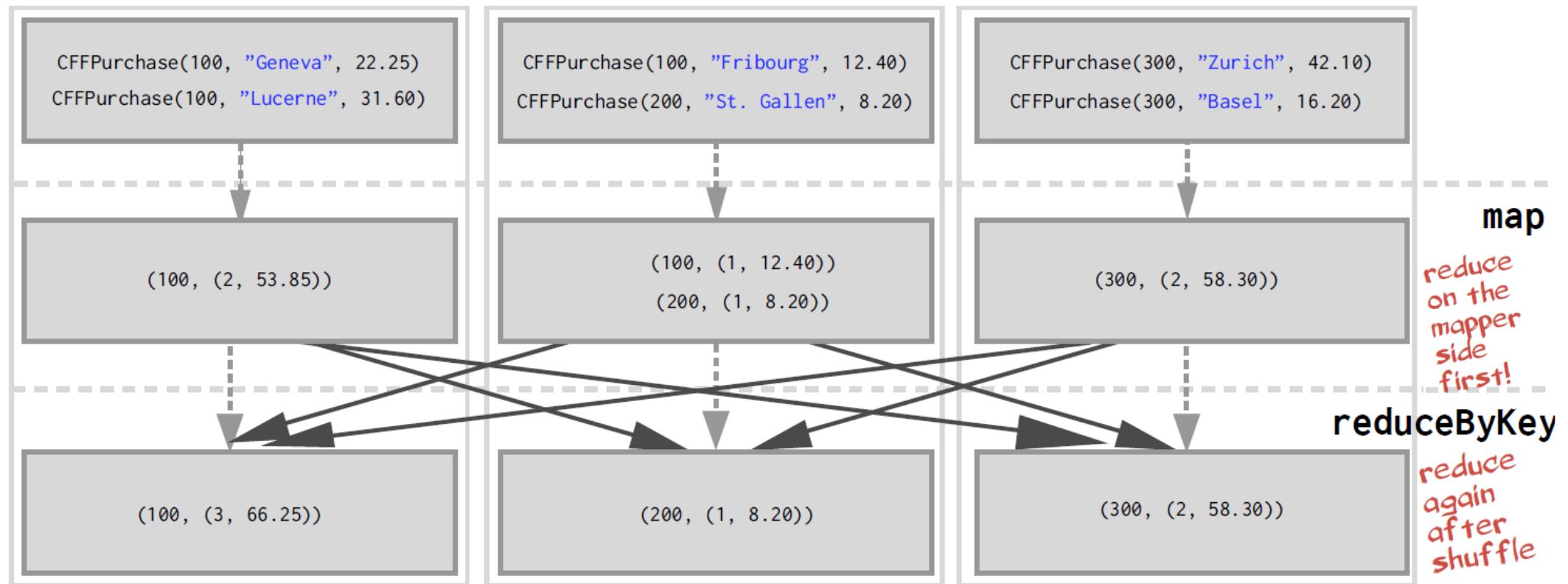
Shuffle Example

```
case class CFFPurchase(customerId: Int, destination: String, price: Double)
```



<https://heather.miller.am/teaching/cs4240/spring2018/pdf/spark-3-1.pdf>

Optimization: Reduce before Shuffle

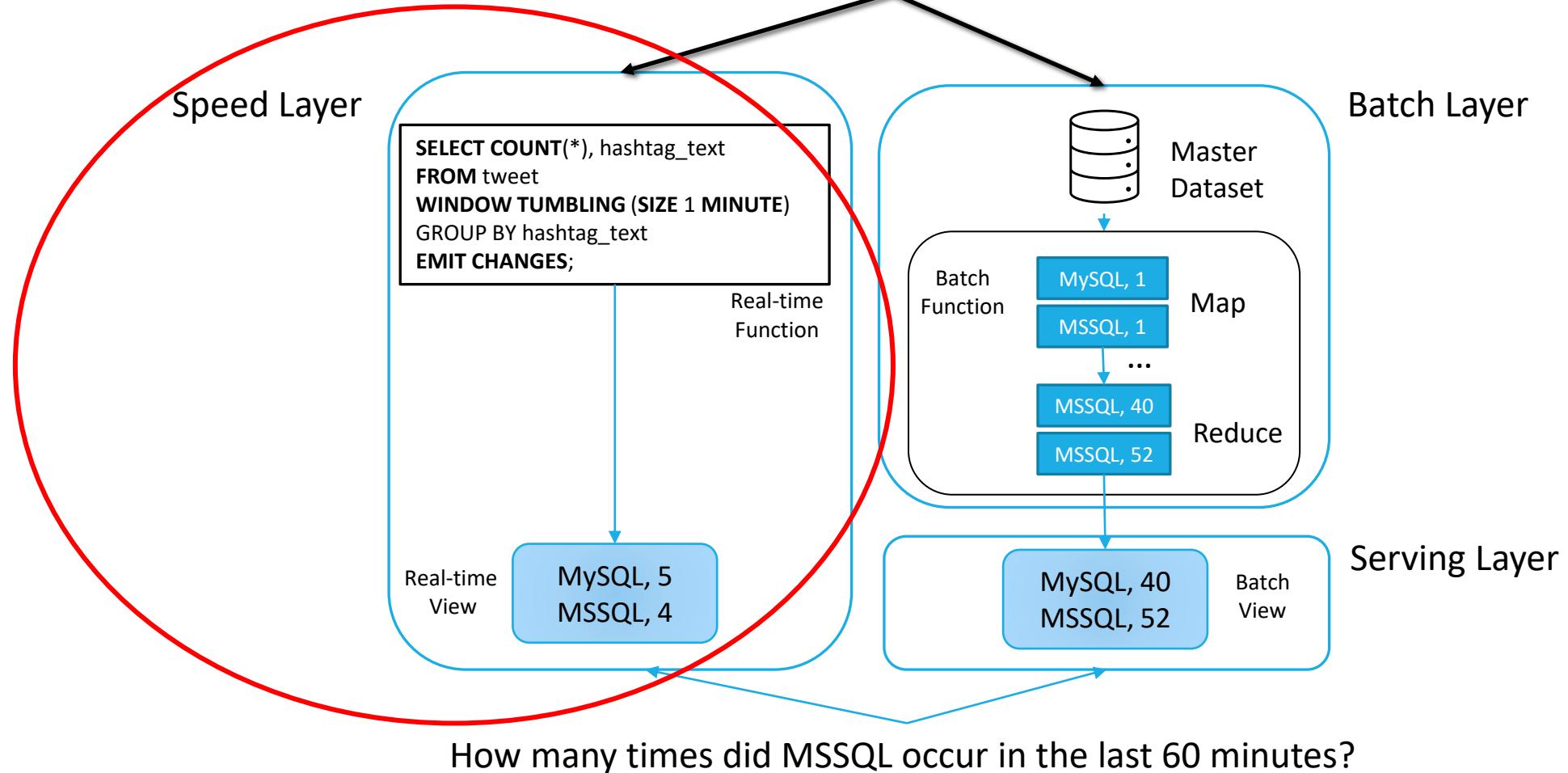
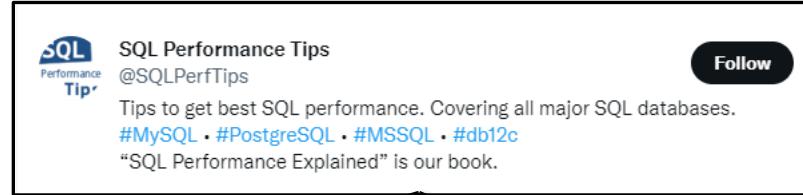




Quiz

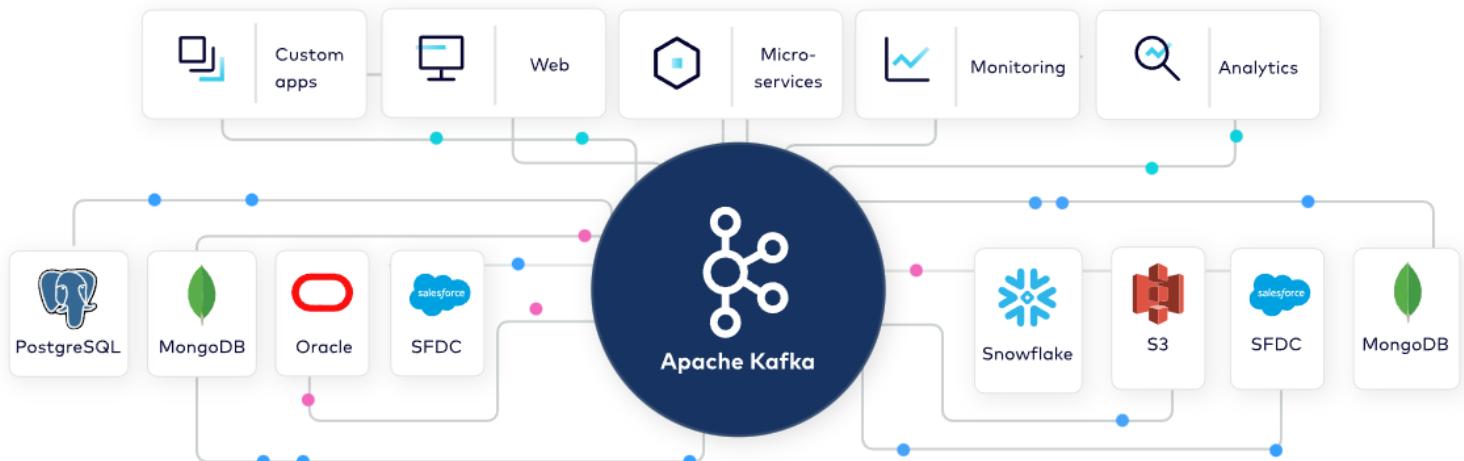
<https://www.menti.com>
Code **4960 2893**





4.2.3 Apache Kafka

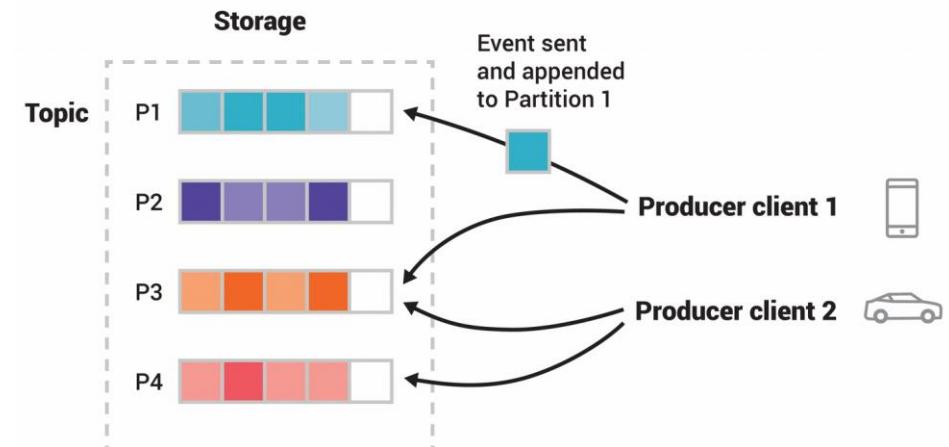
- Distributed streaming platform
 - Publish & subscribe to data streams
 - Process & store streams of events durably
 - Fault-tolerant, distributed, scalable
- „Real-time“ streaming
 - Data pipelines for reliably transferring data between systems
 - Applications that transform or react to the data streams
 - Event-at-a-time processing (not micro-batch)



<https://www.confluent.io/de-de/what-is-apache-kafka/>

Basic Data Model

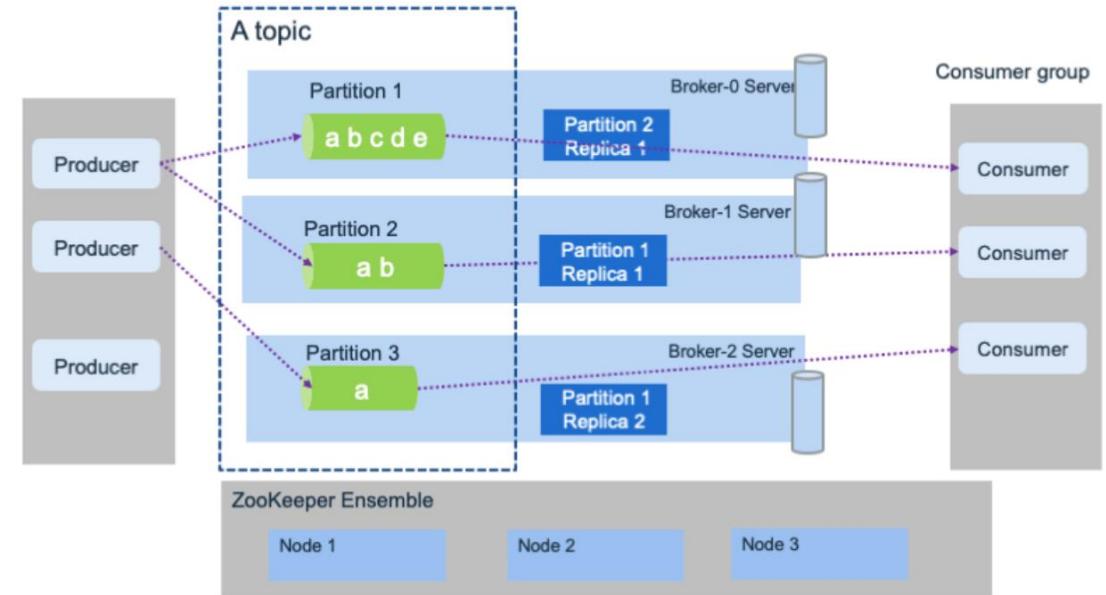
- Event: (key, value, timestamp)
- Topic
 - Organizes and durably stores events
 - Partitioned: topic is spread over buckets on different Kafka brokers
 - Events with same key are written to the same partition



<https://kafka.apache.org/>

Kafka Architecture

- Producer
 - Publishes events to a topic
 - Order of messages only kept p. partition
- Broker
 - Hosts topic log
 - Maintains leader & follower partitions in coord. with Zookeeper
 - Maintains replication of partition across cluster
- Consumer
 - Subscribes to events of a topic
 - Decoupled from producers
- ZooKeeper
 - Keeps track of Kafka cluster nodes status and topics, partitions,..



<https://ibm-cloud-architecture.github.io/refarch-eda/technology/kafka-overview/>

```
KStream<String, Long> onlyPositives =  
    stream.filter( (key, value) -> value > 0 );
```

Kafka Streams API

- Implement stream processing applications and microservices
- Elastic, scalable, fault-tolerant (including state)
- Easy to use and powerful Streams Domain Specific Language (DSL)
 - Record-by-record processing (ms latency)
 - Single message transform (filter, map, etc)
 - Aggregations / Join
 - Windows (time, session)
 - Rich time semantics (event time, ingestion time, processing time)
- Stream-Table duality

KStreams and KTables

- KStream
 - Immutable event stream
 - Each event describes an event in the real world
 - Example: click stream

alice	paris
bob	Zurich
alice	berlin

↓ time

- KTable
 - Changelog *stream*
 - Each record describes a *change* to a previous event
 - Example: position report stream
 - Holds a materialized view of the latest update per key as *internal state*

Example - KTable

Changelog stream

alice	paris
-------	-------

bob	zurich
-----	--------

alice	berlin
-------	--------

KTable state

alice	paris
-------	-------

KTable state

alice	paris
bob	zurich

KTable state

alice	berlin
bob	zurich



Example - Changelog Streams

Record stream

alice	paris
-------	-------

bob	zurich
-----	--------

alice	berlin
-------	--------

count()

KTable state

alice	1
-------	---

count()

KTable state

alice	1
bob	1

count()

KTable state

alice	2
bob	1

Changelog stream (output)

alice	1
-------	---

bob	1
-----	---

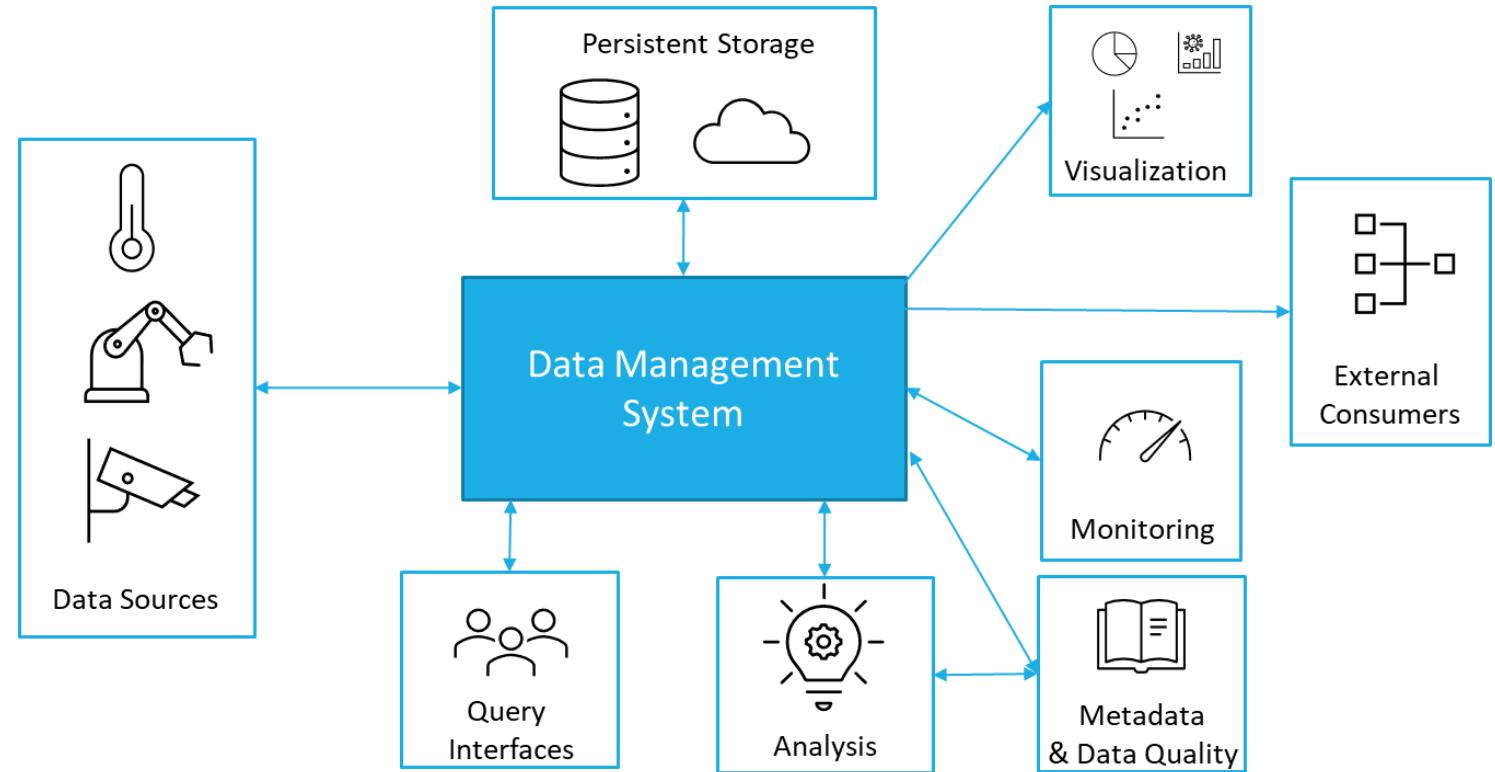
alice	2
-------	---

ksqlDB

- Database which provides an SQL environment for Kafka
- Allows for stream processing and materialized views

```
CREATE STREAM s1 AS
    SELECT COUNT(*), temperature FROM Environment
    WHERE temperature > 20.0
    WINDOW TUMBLING (SIZE 1 MINUTE)
    GROUP BY temperature
    EMIT CHANGES;
```

Lecture Data Stream Management and Analysis





Quiz

<https://www.menti.com>
Code **4960 2893**



Summary

- Pervasive broadband Internet enables data management in the cloud
- Pay-as-you-go computing
- CAP-Theorem: Consistency, Availability, Partitioning → Choose two out of three
- NoSQL DBMS offer new models for data management, featuring
 - Scalability
 - Distributed architectures
 - Schemaless data management
(no a-priori schema definition)
 - New data models (graph, document, ...)
- „NewSQL“ DBMS providing scalable, distributed DBMS functionality and still use SQL and the relational model as a logical data model
- Classical RDBMS providers are catching up with new technologies
- Polyglot persistence
 - Data management today requires a heterogeneous set of data storage technologies, which are optimized for certain applications

Review Questions

- What are the limitations of RDBMS in Internet or Big Data applications?
- Explain the CAP theorem!
- What are the four basic data models for NoSQL DBMS?
- Explain the concepts „Sharding“ and „Eventual Consistency“!
- What is the role of NameNodes and DataNodes in Hadoop?
- How does MapReduce work?
- What are RDDs, partitions, and shuffling in Spark?
- What is polyglot persistence?

References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

- [Agrawal et al., 2011] D. Agrawal et al.: Big Data and Cloud Computing: Current State and Future Opportunities. *Tutorial at EDBT 2011*, <http://www.cs.ucsb.edu/~sudipto/edbt2011/>
- [Banker, 2012] K. Banker: *MongoDB in Action*. Manning, 2012.
Detailed book for MongoDB, addresses also recent features.
- [Bitkom, 2014] Bitkom. Big-Data-Technologien – Wissen für Entscheider – Leitfaden, 2014, <https://www.bitkom.org/sites/default/files/file/import/140228-Big-Data-Technologien-Wissen-fuer-Entscheider.pdf>
- [Boci, E. & Thistletonwaite, S.: A novel big data architecture in support of ADS-B data analytic *Proc. Integrated Communication, Navigation, and Surveillance Conference (ICNS), 2015*, C1-1-C1-8
- [Brewer, 2000] Brewer, E.A.: Towards robust distributed systems (abstract). Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16–19, 2000, Portland, Oregon, USA., 7 (2000)
- [Brewer, 2012] Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. *Computer*, 45(2), 23-29.
- [Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). Fundamentals of database systems (Vol. 7). Pearson
- [Lourenço et al., 2015] Lourenço, J. R., Cabral, B., Carreiro, P., Vieira, M., & Bernardino, J. (2015). Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1), 1-26.

References & Further Reading (2)

- [McMurtry et al., 2013] D. McMurtry et al.: Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence. Microsoft, 2013. Good overview of different NoSQL data models, available online: <https://www.microsoft.com/en-us/download/details.aspx?id=40327>
- [Marz & Warren, 2015] Nathan Marz, James Warren, Big Data: Principles and best practices of scalable real-time data systems, Manning, 2015
- [Rahm & Sattler, 2015] Rahm, E., Saake, G., & Sattler, K. U. (2015). Verteiltes und paralleles Datenmanagement. Springer Berlin Heidelberg.
- [Raj & Deka, 2018] Pethuru Raj and Ganesh Chandra Deka. *A Deep Dive into NoSQL Databases: the Use Cases and Applications*, Elsevier Science & Technology, 2018.
- [Redmond & Wilson, 2012] E. Redmond, J.R. Wilson: Seven Databases in Seven Weeks. Pragmatic Programmers, 2012. Hands-on book for seven important DBMS, many technical details, less suited for an overview.
- [Sadalage & Fowler, 2021] P.J. Sadalage, M. Fowler: NoSQL Distilled. Addison-Wesley, 2012. Good overview of the key concepts of NoSQL systems; has a system independent and a system specific part.
- [Thusoo et al., 2010] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, Hao Liu: Data warehousing and analytics infrastructure at facebook. SIGMOD Conference 2010: 1013-1020

Implementation of Databases

Chapter 5: Advanced Query Processing

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

Excercises

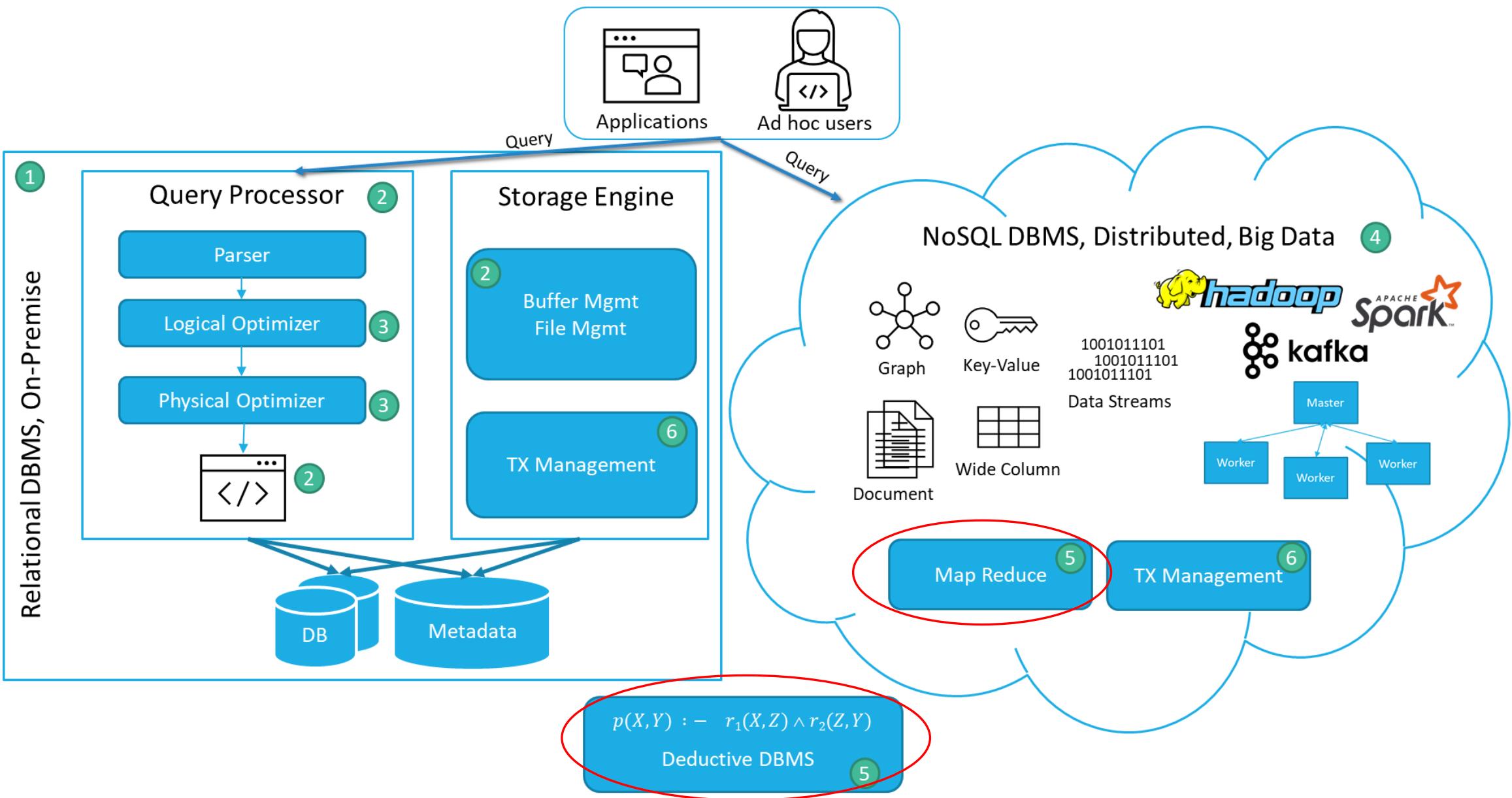
Anastasiia Belova, M.Sc.

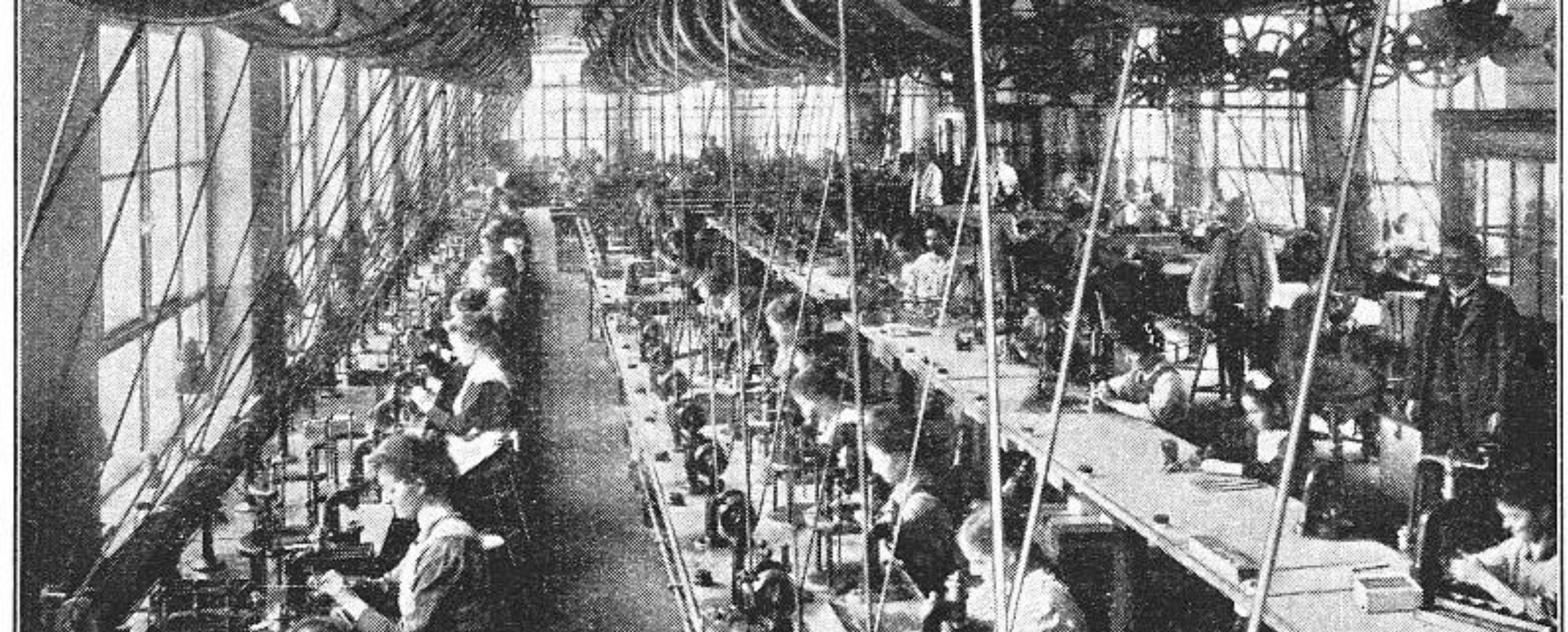
Soo-Yon Kim, M.Sc.



Juniorprofessur
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY





5.1 Query Processing in Big Data Systems

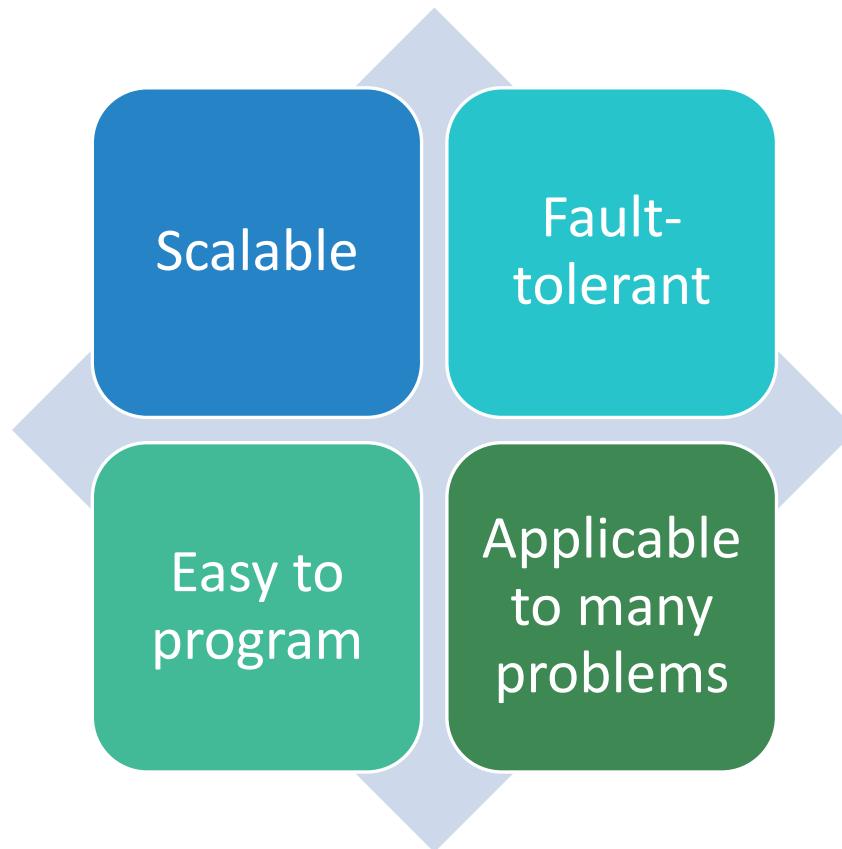
Learning Goals

At the end of this section you will be able to

- ✓ implement a MapReduce job
- ✓ explain key concepts & limitations of MapReduce
- ✓ name and explain higher-level languages for MapReduce jobs
- ✓ define a query in Apache Pig, Apache Hive, Apache Spark
- ✓ name four methods how to use SQL on Hadoop

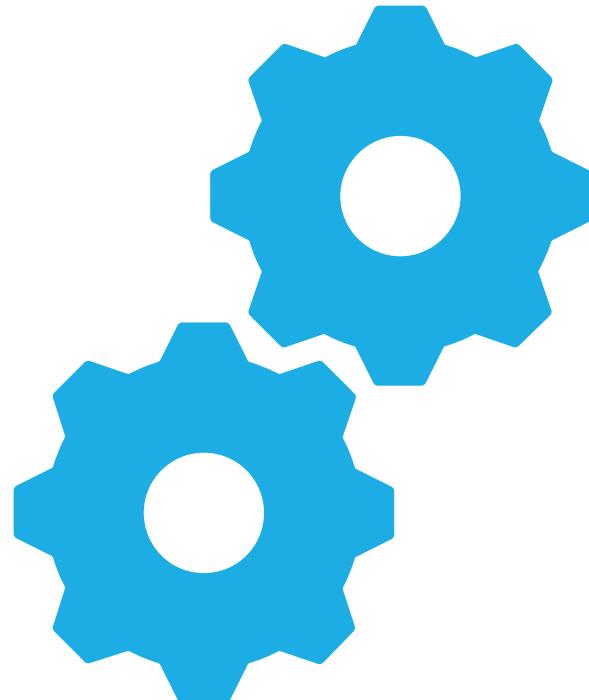


Requirements for a Distributed System



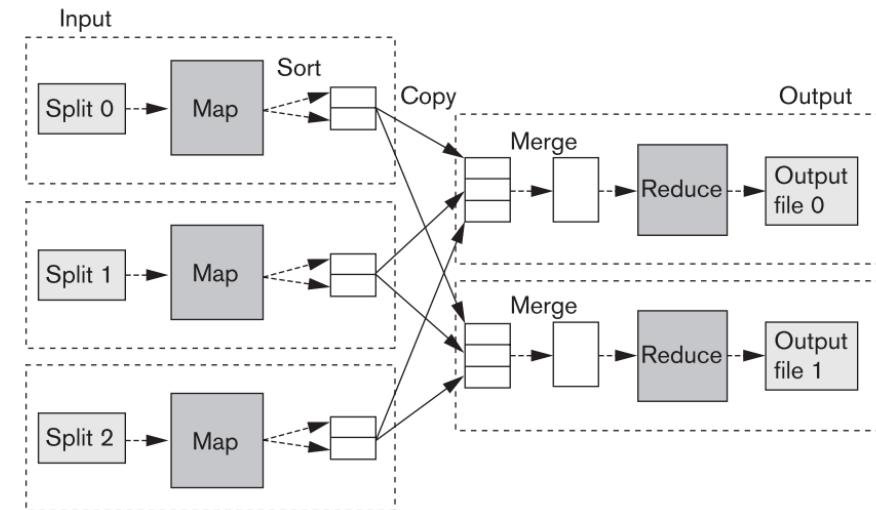
Challenges in Distributed Programming

- Lots of programming work
 - Communication and coordination
 - Work partitioning
 - Status reporting
 - Optimization
 - Locality
- Duplicated for every problem you want to solve
- Needs to be fault-tolerant
 - One server may stay up three years (1,000 days)
 - If you have 10,000 servers, expect to lose 10 a day



MapReduce

- Programming pattern for parallel computation in distributed system
- **Function Map(data) → (key, value)**
 - Reads input data and emits key-value pairs
 - Keys are not necessarily unique in emitted pairs
- **Function Reduce(key, values) → (key, values)**
 - Gets input from Map function - set of values for a single key
 - Input and output structure should be the same
- Map and Reduce jobs can run on different nodes



[Elmasri & Navathe, 2017]

Example: SQL Schema & Queries

CartItem

ID	QTY	PID

Sum of all items in shopping carts, grouped by product ID

```
SELECT PID, SUM(QTY)  
FROM CartItem  
GROUP BY PID
```

Product

ID	Name	Price

Sum of the value of all items in shopping carts, grouped by product ID

```
SELECT PID, SUM(QTY)*Price  
FROM CartItem, Product P  
WHERE P.ID=PID  
GROUP BY PID
```

Simple Example for MapReduce

(in MongoDB syntax)

Query: Sum of items in customers' shopping carts, grouped by product ID

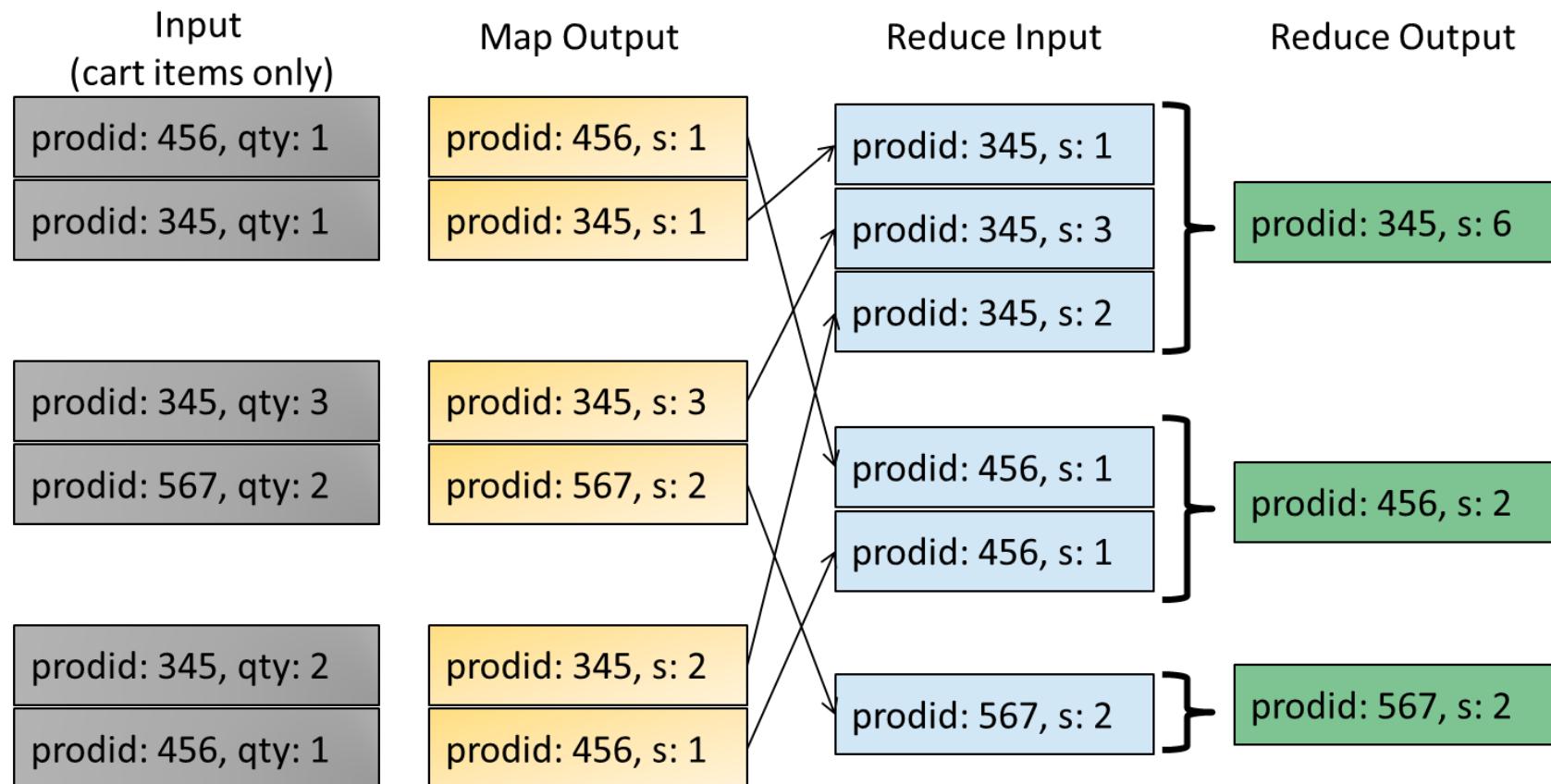
Input: Customer collection

```
{  
    firstname: "John",  
    lastname: "Doe",  
    address: { ... },  
    cart: [ { prodid: 3456,  
              qty: 2},  
            { prodid: 6789,  
              qty: 1} ]  
}
```

```
map=function() {  
    if(this.cart!=null) {  
        this.cart.forEach(function(item) {  
            emit(item.prodid, { sum: item.qty });  
        });  
    }  
}
```

```
reduce=function(key, values) {  
    var s=0;  
    values.forEach(function(value) {  
        s+=value.sum;  
    });  
    return ( { sum: s} );  
}
```

Example



A More Complex Example (1)

Join between customers and products

Input: Customers + products collection

```
{  
    firstname: "John",  
    lastname: "Doe",  
    address: { ... },  
    cart: [ { prodid: 3456,  
              qty: 2},  
            { prodid: 6789,  
              qty: 1} ]  
}  
  
{  
    prodid: 3456,  
    price: 34  
}  
  
{  
    prodid: 6789,  
    price: 23  
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
map=function() {  
    if(this.cart!=null) {  
        this.cart.forEach(function(item) {  
            emit(item.prodid, { qty: item.qty,  
                               price: null,  
                               total: null });  
        });  
    }  
    if(this.prodid!=null) {  
        emit(this.prodid, {qty: 0,  
                          price: this.price,  
                          total: 0});  
    }  
}
```

A More Complex Example (2)

Join between customers and products

Input: Customers + products collection

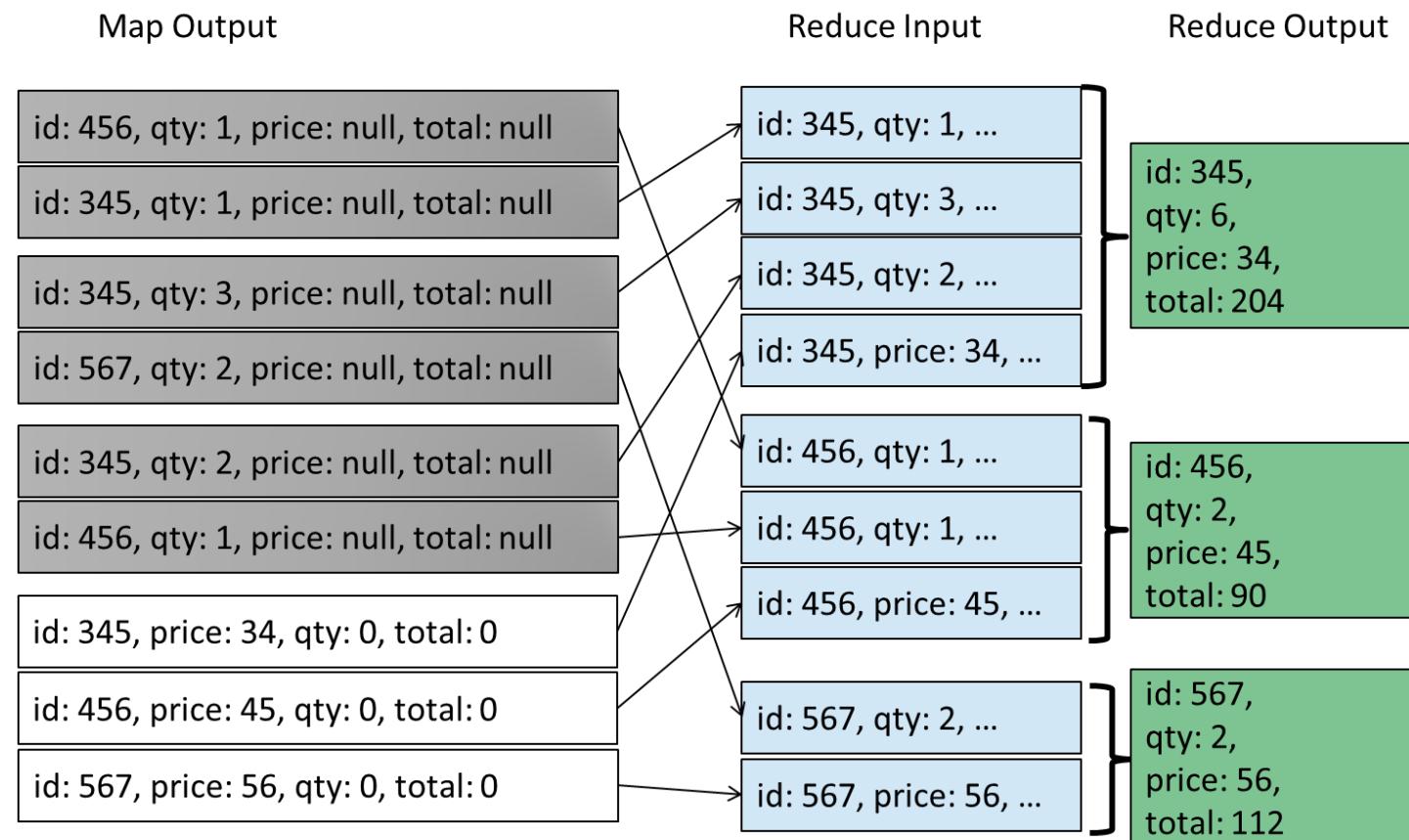
```
{  
    firstname: "John",  
    lastname: "Doe",  
    address: { ... },  
    cart: [ { prodid: 3456,  
              qty: 2},  
            { prodid: 6789,  
              qty: 1} ]  
}  
  
{  
    prodid: 3456,  
    price: 34  
}  
  
{  
    prodid: 6789,  
    price: 23  
}
```

Query: The sum of items and their value in all shopping carts, grouped by product ID

```
reduce=function(key, values) {  
    var res = { qty : 0, price: null, total: null };  
    values.forEach(function(value) {  
        res.qty+=value.qty;  
  
        if(res.price!=null) {  
            res.total+=res.price*value.qty;  
        }  
  
        if(res.price==null && value.price!=null) {  
            res.price=value.price;  
            res.total=res.price * res.qty;  
        }  
    });  
    return res;  
}
```

A More Complex Example (3)

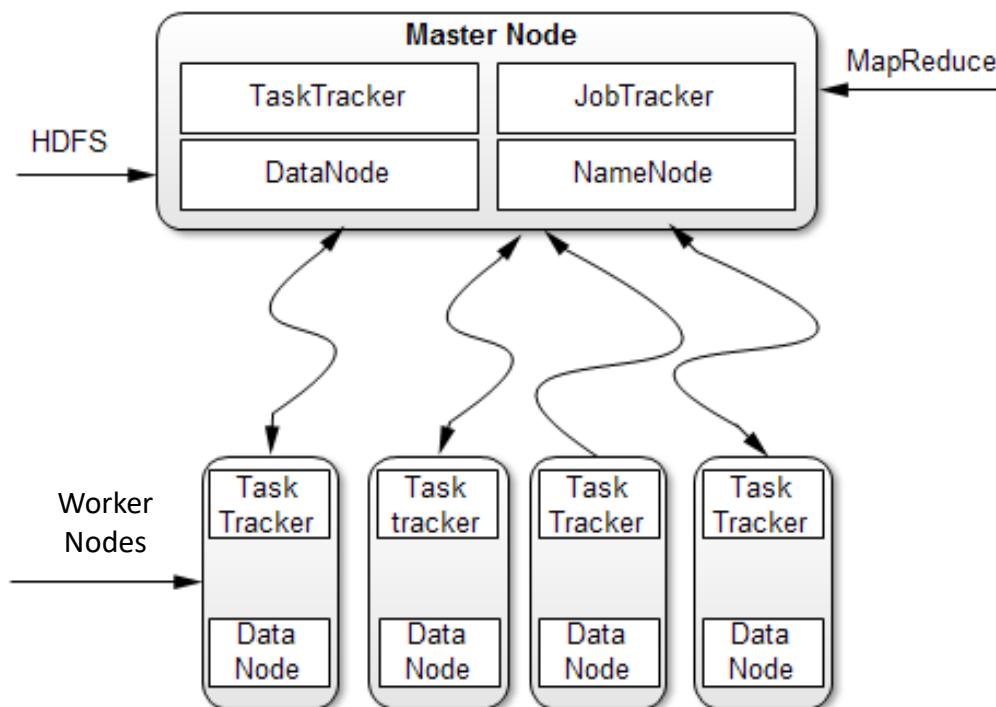
Join between customers and products



Key Points of MapReduce

- Many implementations, e.g., in Hadoop with HDFS or NoSQL systems with database query processing
- No coordination is required between individual map and reduce jobs
- Sharding & replication fits nicely into MapReduce pattern
 - Parallelism is increased: Map tasks can be distributed to different shards or their replicas
 - Availability is increased: if a node is not available, replica can take over the job
- Mapping of input data is important
 - Key determines grouping
 - Data structure of value is output

Fault-Tolerance



- Worker failure
 - Detect failure via periodic heartbeats
 - Re-execute completed and in-progress map tasks
 - Re-execute in progress reduce tasks
 - Task completion committed through master
- Master failure
 - Is much more rare
 - Fail-over to secondary master nodes (e.g., name node, resource manager)

MapReduce Limitations

- Many queries/computations need multiple MR jobs
- 2-stage computation too rigid
- Example: Find the top 10 most visited pages in each category

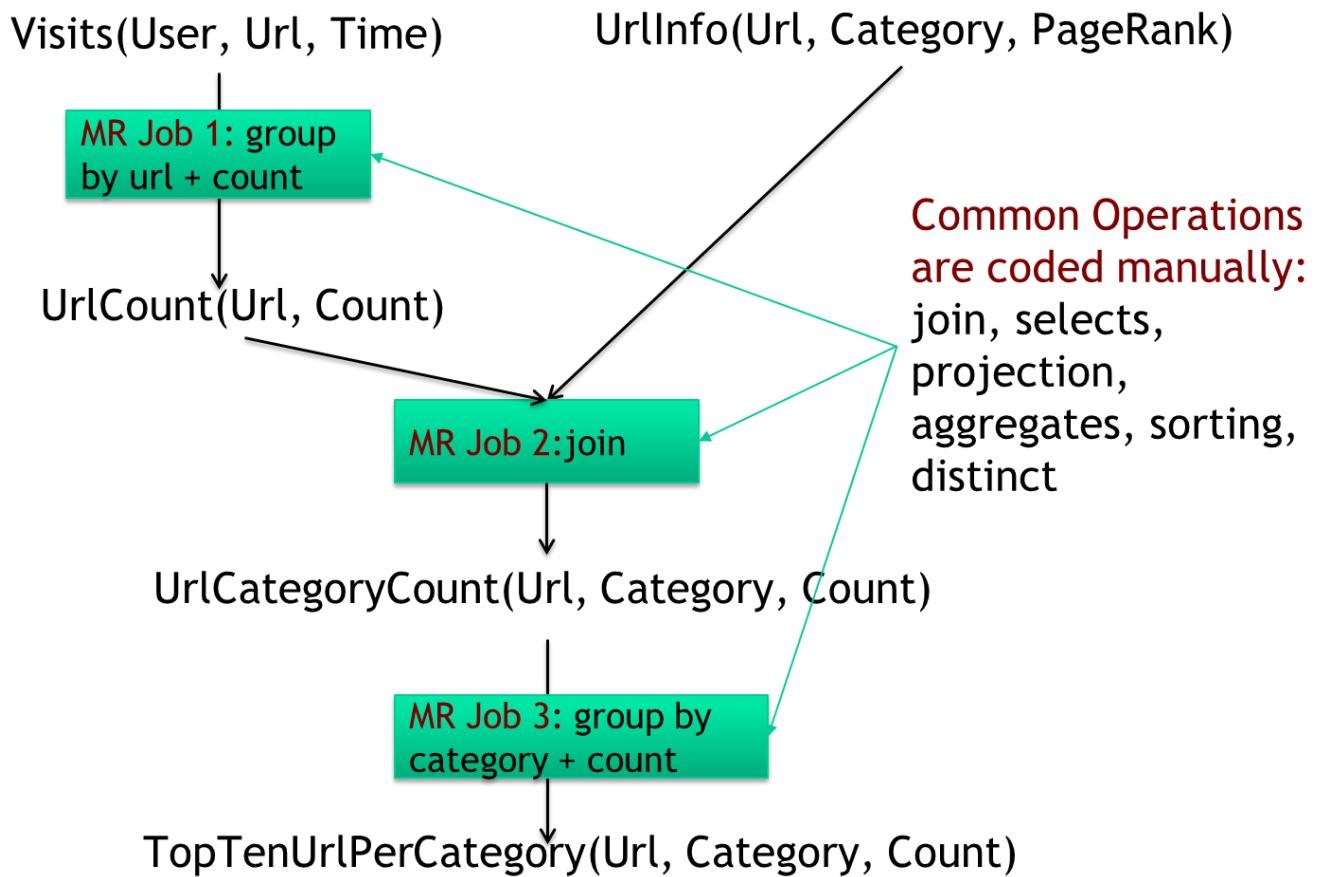
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

UrlInfo

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Example: Top 10 Most Visited Pages per Category



MapReduce is not a Good Model for Data Processing

- Required
 - Support for algebra operations: join, selection, projection, group by, ...
 - More abstract language to implement data transformations and queries

- High-level languages that are compiled into MapReduce jobs
 - Apache Pig (Pig Latin)
 - Apache Hive (HiveQL)
 - Spark SQL



Apache Pig

- High-level language for analyzing large data sets
- Apache project since 2007, mainly supported by Twitter and Hortonworks
- Pig Latin: Procedural language with algebra-like operations (join, selection, projection, ...)
- Workflows can be defined as step-by-step procedural scripts
- Pig Latin can be compiled to jobs on
 - Hadoop
 - Spark
 - Tez

Based on: [Olston et al., 2008]

Pig Latin Example of Top 10 Visited Pages

Operates Directly Over Files

```
visits          = load '/data/visits' as (user, url, time);  
gVisits        = group visits by url;  
urlCounts      = foreach gVisits generate url, count(visits);  
  
urlInfo         = load '/data/urlInfo' as (url, category, pRank);  
urlCategoryCount = join urlCounts by url, urlInfo by url;  
gCategories    = group urlCategoryCount by category;  
  
topUrls        = foreach gCategories generate top(urlCounts,10);  
  
store topUrls into '/data/topUrls';
```

Optional Schema & Schema-on-read

```
visits          = load '/data/visits' as (user, url, time);  
gVisits        = group visits by url;  
urlCounts      = foreach gVisits generate url, count(visits);  
  
urlInfo         = load '/data/urlInfo' as (url, category, pRank);  
urlCategoryCount = join urlCounts by url, urlInfo by url;  
gCategories    = group urlCategoryCount by category;  
  
topUrls        = foreach gCategories generate top(urlCounts,10);  
  
store topUrls into '/data/topUrls';
```

User-defined Functions (UDFs)

```
visits           = load '/data/visits' as (user, url, time);  
gVisits         = group visits by url;  
urlCounts       = foreach gVisits generate url, count(visits);  
  
urlInfo          = load '/data/urlInfo' as (url, category, pRank);  
urlCategoryCount = join urlCounts by url, urlInfo by url;  
gCategories     = group urlCategoryCount by category;  
  
topUrls          = foreach gCategories generate top(urlCounts,10);  
  
store topUrls into '/data/topUrls';
```

Can be used in every construct of
▪ Load, Store
▪ Group, Filter, Foreach

Example: Compile to MapReduce Jobs

```
visits           = load '/data/visits' as (user, url, time);  
gVisits         = group visits by url;  
urlCounts       = foreach gVisits generate url, count(visits);
```

MR Job 1

```
urlInfo          = load '/data/urlInfo' as (url, category, pRank);  
urlCategoryCount = join urlCounts by url, urlInfo by url;
```

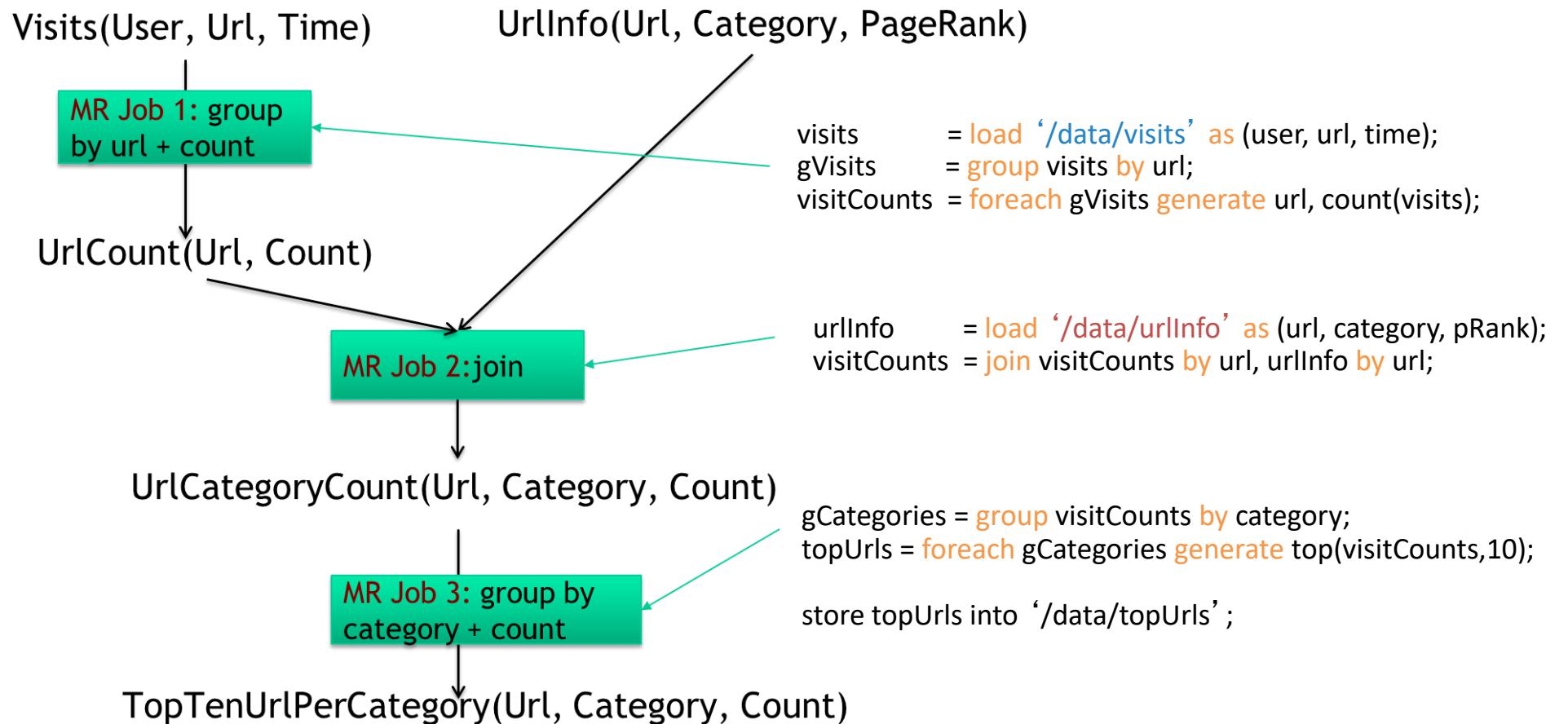
MR Job 2

```
gCategories      = group urlCategoryCount by category;  
topUrls          = foreach gCategories generate top(urlCounts, 10);
```

MR Job 3

```
store topUrls into '/data/topUrls';
```

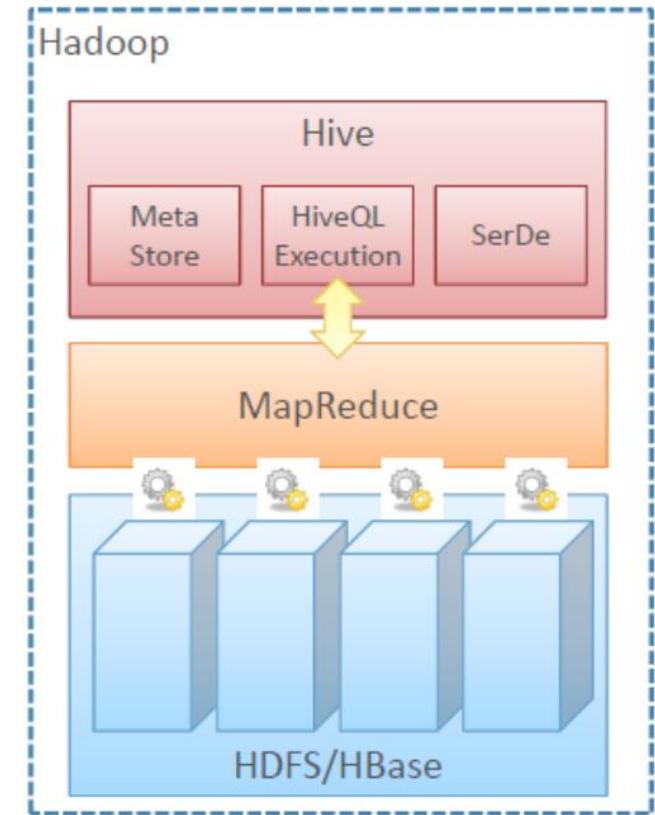
Example: Compile to MapReduce Jobs





Apache Hive

- Data warehouse software for large datasets in distributed storage
- Mature part of every Hadoop distribution
- **Hive-QL**
 - SQL-like declarative language with UDFs extendable
e.g., SELECT *, INSERT INTO, GROUP BY, SORT BY
- Table definitions in Hive Meta Store
- Schema-on-Read via SerDe with many data formats
- Compiles to Hadoop, Spark, Tez
- Batch-oriented, slow



Based on: [Thusoo et al., 2009]

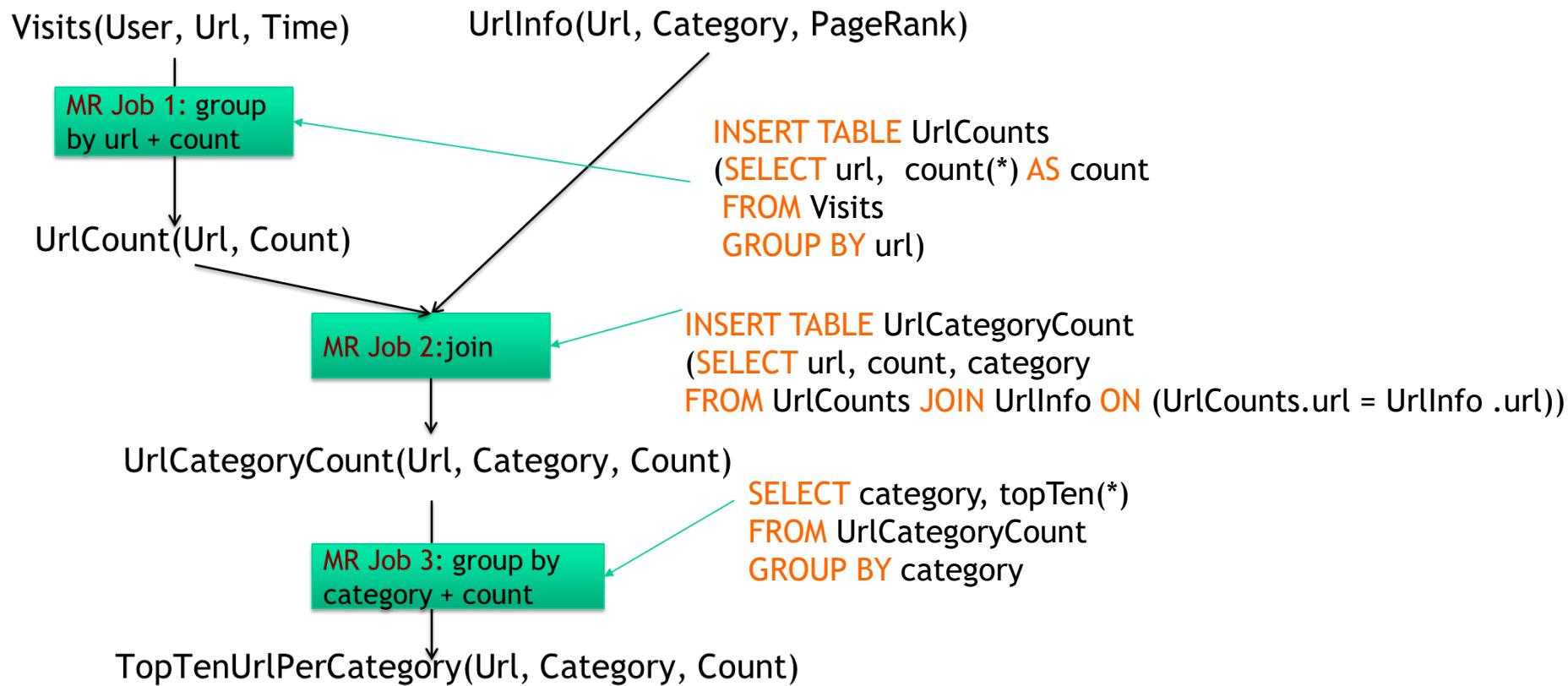
Hive Example

```
INSERT TABLE UrlCounts  
(SELECT url, count(*) AS count  
FROM Visits  
GROUP BY url)
```

```
SELECT category, topTen(*)  
FROM UrlCategoryCount  
GROUP BY category
```

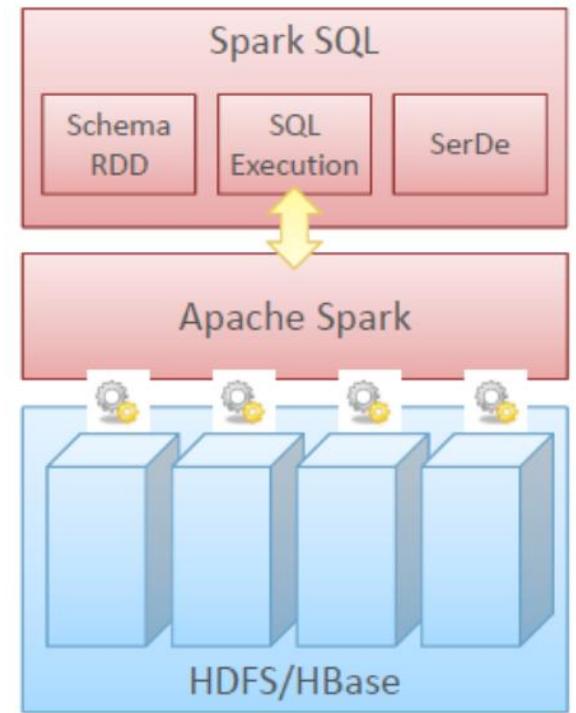
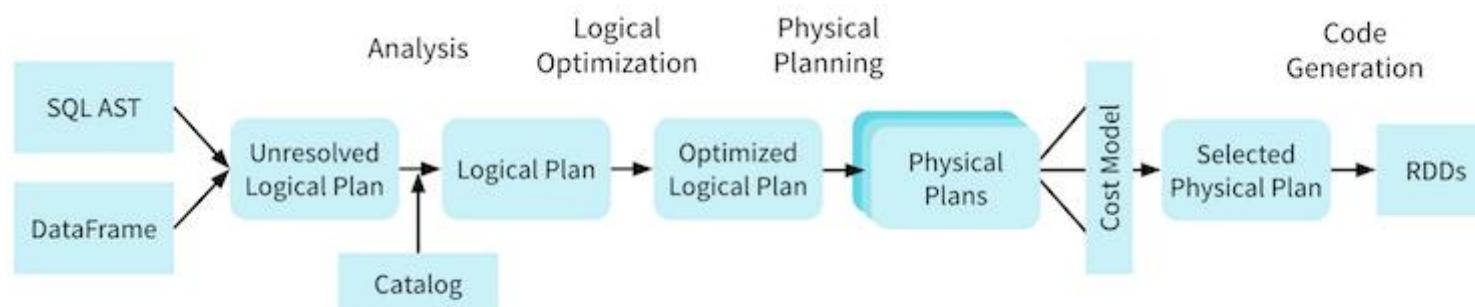
```
INSERT TABLE UrlCategoryCount  
(SELECT url, count, category  
FROM UrlCounts JOIN UrlInfo ON (UrlCounts.url = UrlInfo.url))
```

Example: Compile to MapReduce Jobs



Apache Spark (see also Chapter 4)

- Queries can be also specified in Spark SQL (SQL dialect of Apache Spark)
- DataFrames represent relational tables, implemented as RDDs
- Query processing are transformations & actions on RDDs/DataFrames
- In-memory columnar format, HDFS / Hbase as file format
- Supports Hive extensions, such as UDFs, SerDes, Hive metadata



Apache Spark Example

```
visits = spark.read.format("com.databricks.spark.csv")  
        .option("header","true").load("visits.csv")
```

```
urlinfo = spark.read.format("com.databricks.spark.csv")  
        .option("header","true").load("urlinfo.csv")
```

```
visits.createOrReplaceTempView("visits")  
urlinfo.createOrReplaceTempView("urlinfo")
```

Create tables for
SQL queries

```
q = spark.sql("SELECT v.url, u.category, count(*)  
              FROM visits v, urlinfo u  
              WHERE v.url=u.url  
              GROUP BY v.url, u.category")
```

```
q.explain()
```

Show query plan
(with True as value
shows also the logical plan)

Filtering of top 10 URLs by
category has to be done
separately

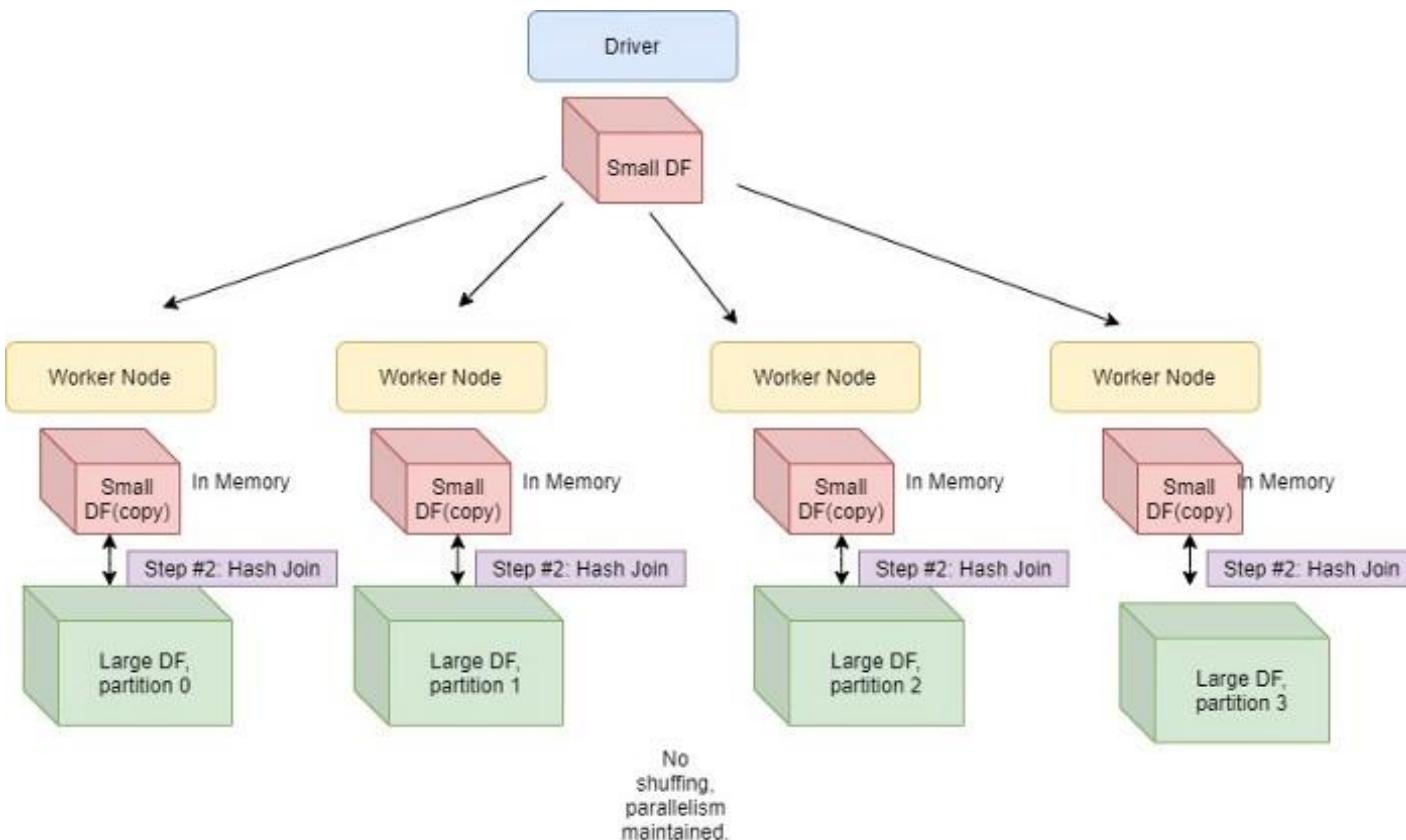
Apache Spark Query Plan

```
scala> q.explain()
== Physical Plan ==
*HashAggregate(keys=[url#13, category#40], functions=[count(1)])
+- Exchange hashpartitioning(url#13, category#40, 200)
   +- *HashAggregate(keys=[url#13, category#40], functions=[partial_count(1)])
      +- *Project [url#13, category#40]
         +- *BroadcastHashJoin [url#13], [url#39], Inner, BuildRight
            :- *Project [url#13]
            :  +- *Filter isnotnull(url#13)
            :    +- *FileScan csv [url#13] Batched: false, Format: CSV, Location: InMemoryFileIndex
[file:/home/scala/visits.csv], PartitionFilters: [], PushedFilters: [IsNotNull(url)], ReadSchema: st
ruct<url:string>
            +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
               +- *Project [url#39, category#40]
                  +- *Filter isnotnull(url#39)
                     +- *FileScan csv [url#39,category#40] Batched: false, Format: CSV, Location: In
MemoryFileIndex[file:/home/scala/urlinfo.csv], PartitionFilters: [], PushedFilters: [IsNotNull(url)]
, ReadSchema: struct<url:string,category:string>

scala> _
```

Broadcast operations → Shuffle data between nodes in cluster

Broadcast Hash Join



- Traditional joins are expensive as data is split
- Copy of smaller RDD is sent to each worker node
- In the best case, this RDD fits into memory
- Retain only those columns of the RDD necessary for join
- Uses BitTorrent p2p protocol to broadcast

<https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf>

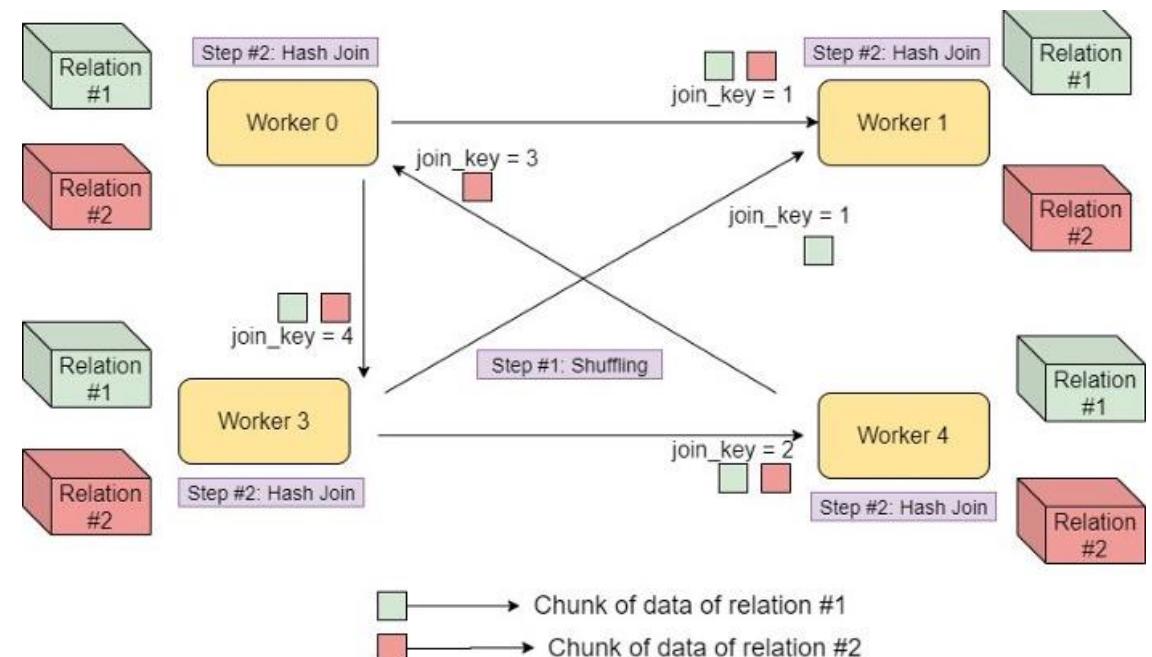
Shuffle Hash Join

Remember

- Data objects are assigned to a partition
 - Hash, range, or custom partitioners available

Cases

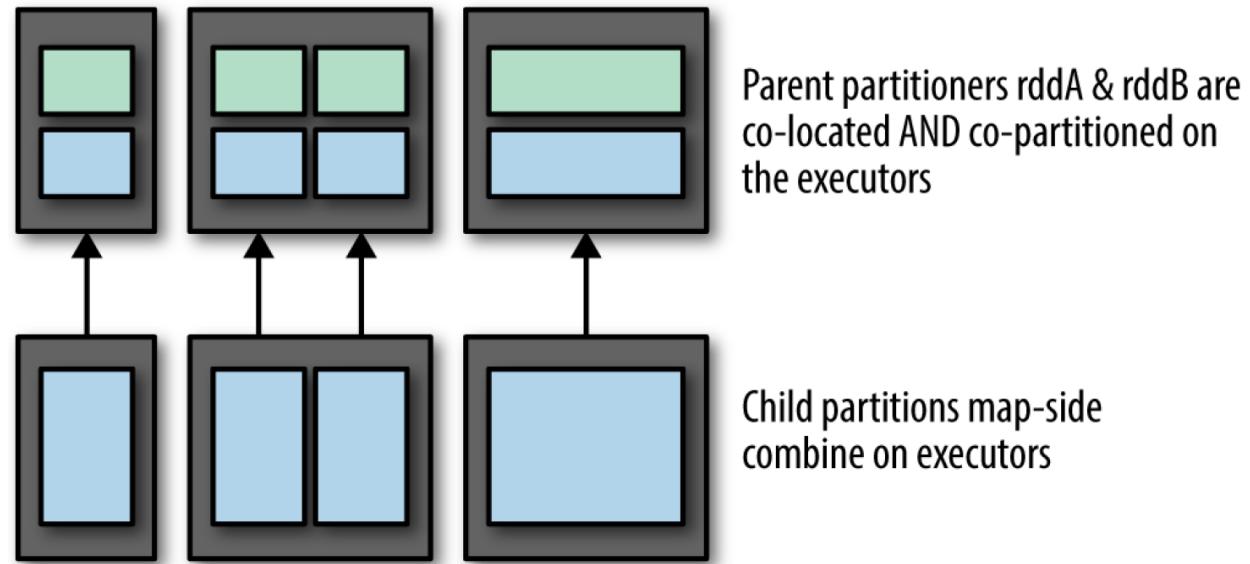
1. Both RDDs use different or custom partitioners (semantics is unknown),
→ shuffle (partition again) with common partitioner
2. One known partitioner → only one RDD needs to be shuffled with the other partitioner
3. Both RDDs have the same partitioner
→ co-located join



<https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf>

Co-located Join

- If both RDDs have the same partitioner, then the partitions to be joined are on the same node and join is executed on these nodes
- Spark has rules how to pick join strategy based on hints, RDD sizes, sortability of key, join type



[Karau & Warren, 2017]



Quiz

<https://www.menti.com>
Code **5517 2837**



Why SQL on Hadoop?

[Albrecht, 2018]

SQL

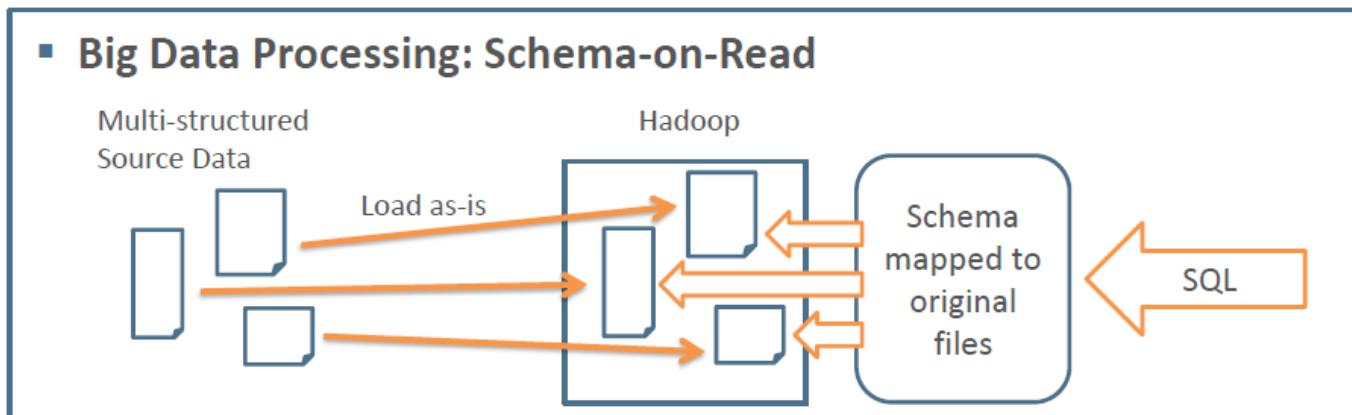
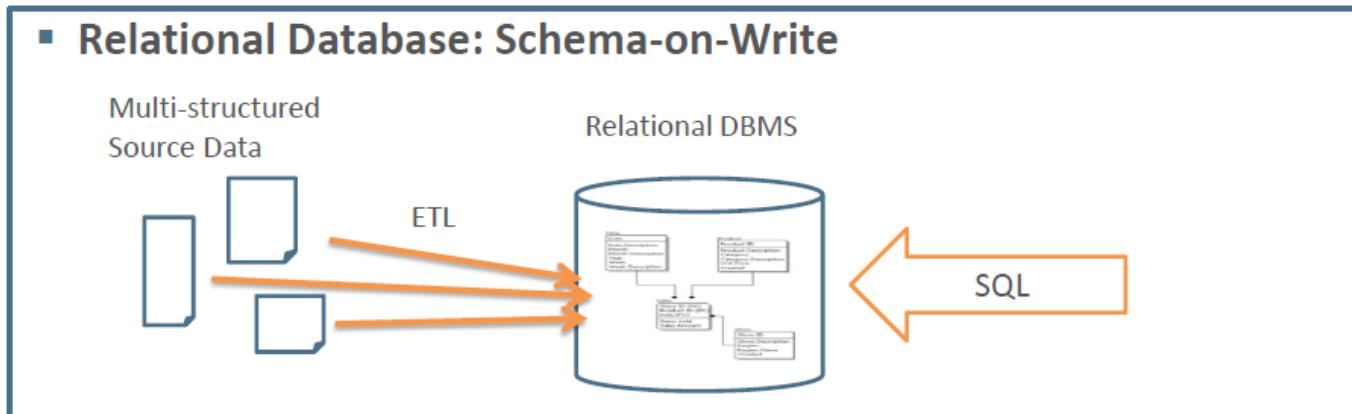
- Mature technology
- Broad knowledge available
- Powerful query language
- High interactive performance
- Many 3rd party tools for data analysis and visualization

Hadoop

- Flexible data structures: semi-structured data, changing schemas
- Self Service: Data integration on-the-fly
- Scalability
- (Relatively) low costs: runs on commodity hardware, open source

Schema-on-Write vs. Schema-on-Read

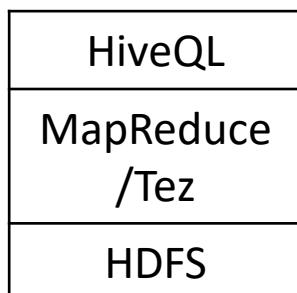
[Albrecht, 2018]



SQL on Hadoop

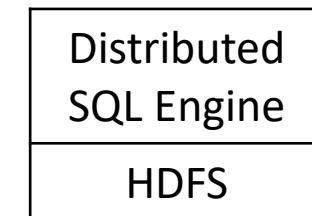
[Albrecht, 2018]

Hive (Native Hadoop)



Stinger
Stinger.next

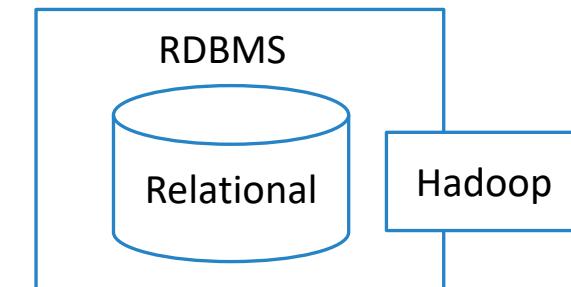
Pure Hadoop SQL Engines



Format-agnostic SQL Engines

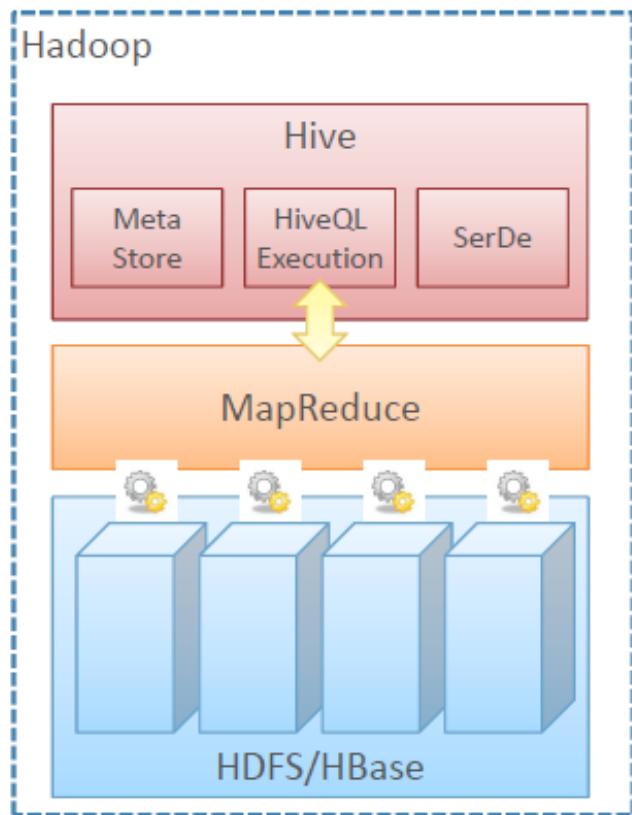


RDBMS with Hadoop Access



Hive

[Albrecht, 2018]



■ General

- SQL-processing for HDFS and HBase
- Table definitions in Hive Metadata Store
- Generation of MapReduce code
- Schema-on-read via SerDe

■ Advantages

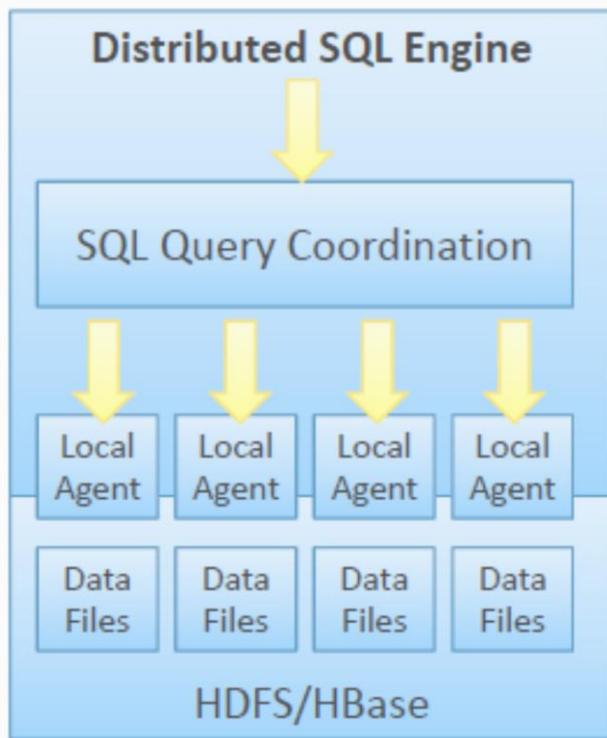
- Mature part of Hadoop
- Simple setup
- Java-API for UDFs
- Usage of many data formats via SerDe

■ Disadvantages

- Batch-oriented, slow

Pure Hadoop SQL Engines

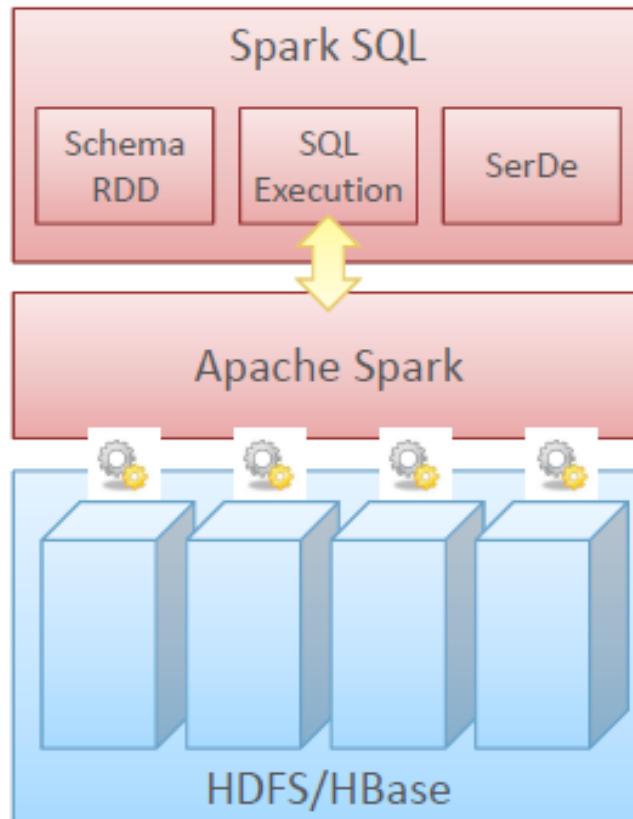
[Albrecht, 2018]



- Approach
 - Distributed parallel SQL engine
 - Often usage of Hive Metadata
 - Support of optimized data formats (Avro, Parquet,...)
 - Hadoop as mandatory basis
- Advantages
 - Significantly faster than Hive
 - Low latency through dedicated engine
 - Operator pipelining and result caching
- Solutions differ in
 - Supported SQL functionality
 - Point querying
 - Cost-based optimizer /performance
 - Transaction support

Apache Spark & Spark SQL

[Albrecht, 2018]



■ General

- SQL engine based on Spark
- Data access via data frames
- In-memory columnar format
- HDFS /HBase as file format

■ Advantages

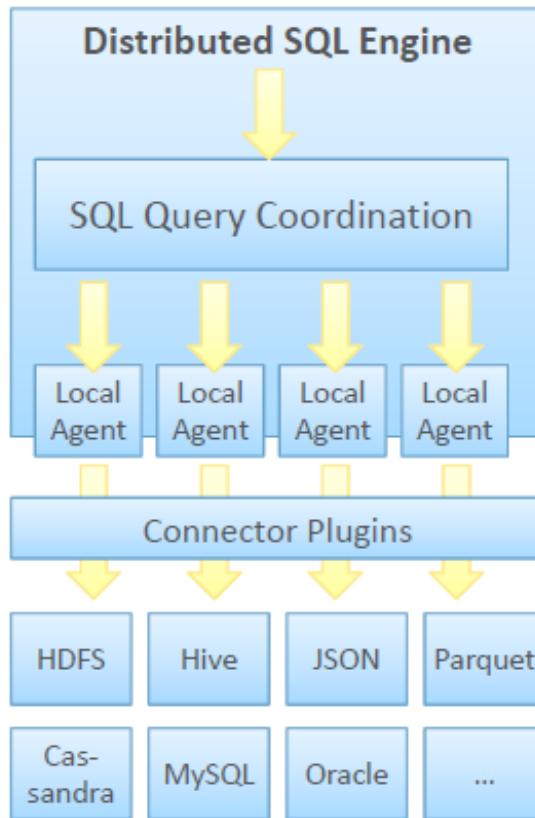
- Spark as general-purpose parallel computing framework
- Support of Hive extensions like UDFs and SerDes and Hive Metadata

■ Disadvantages

- Not yet fully mature
- Not yet as fast as competitors

SQL-Engine with Pluggable Storage

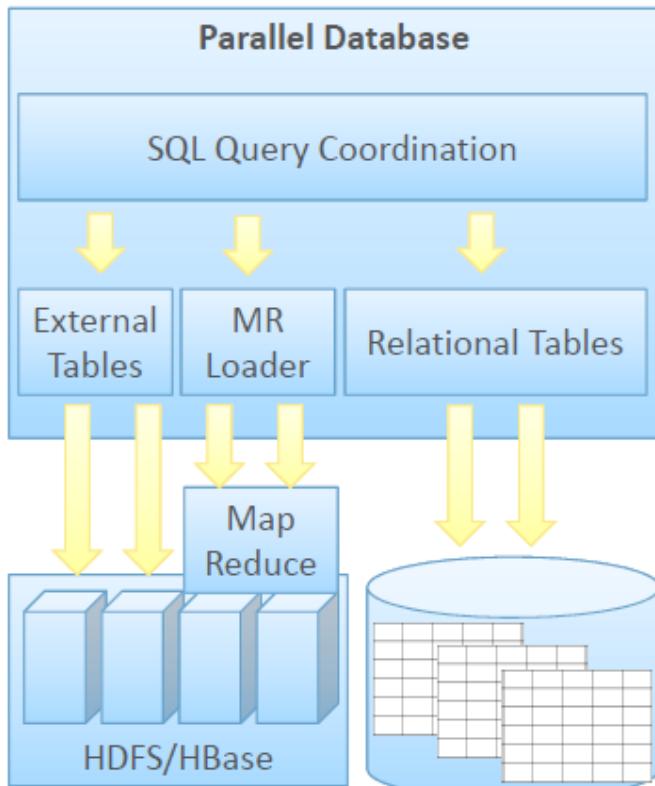
[Albrecht, 2018]



- Approach
 - Distributed, parallel SQL Engine
 - Often uses Hive Metadata
 - Support for optimized data formats
 - Hadoop obligatory as base system
- Advantages
 - Faster than Hive
 - Low latency as Map-Reduce is avoided
 - Pipeling & Caching
 - Scalability
- Disadvantages
 - Usually no transaction management

RDBMS with Hadoop Integration

[Albrecht, 2018]



■ Approach

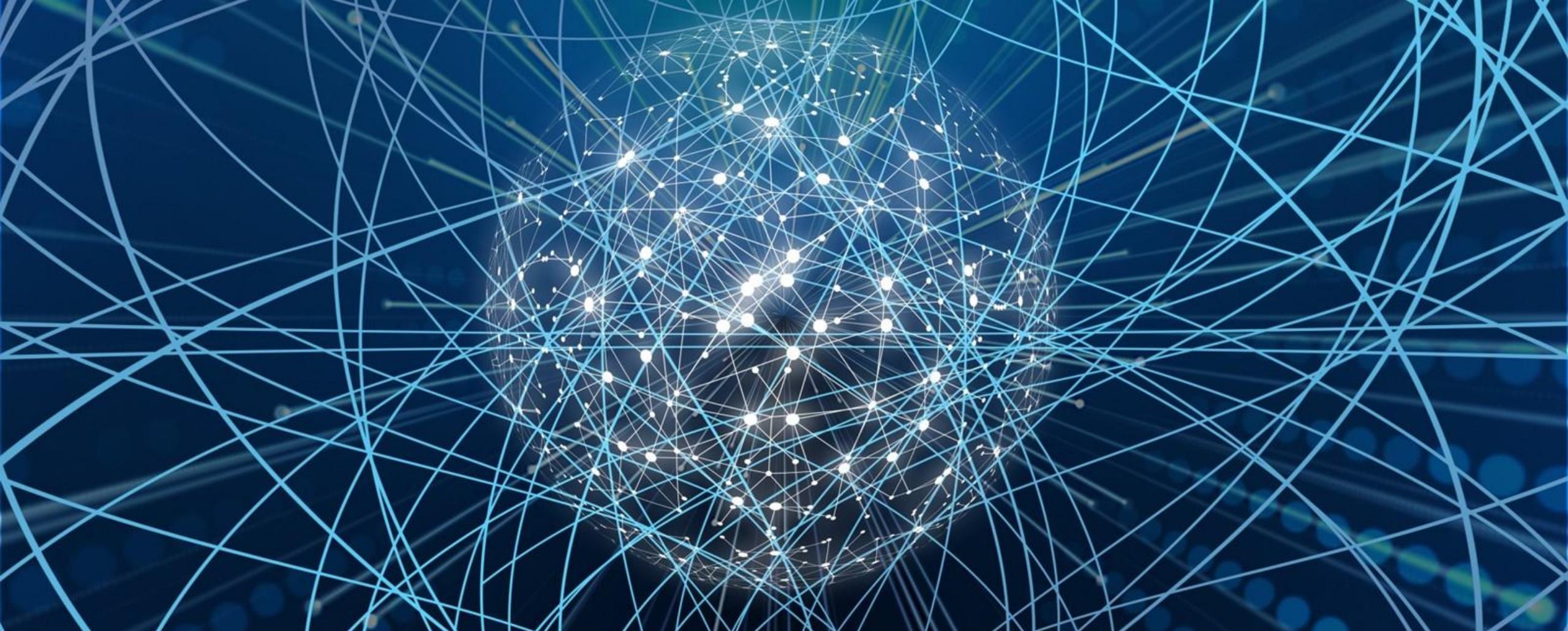
- Towards genuine integration of Hadoop into RDBMS
- Utilize Hadoop's computational power

■ Advantages

- Easiest way to use Hadoop as data source
- Combined access to traditional and new data sources

■ Disadvantages

- Cost
- Limited data sources
- Vendor lock-in



5.2 Deductive Databases

Learning Goals

At the end of this section you will be able to

- ✓ name the components of a deductive database
- ✓ write down facts, rules, constraints, and queries for a database
- ✓ distinguish sublanguages of Datalog
- ✓ create the Herbrand Base and models for a given set of facts and rules
- ✓ do a least fixpoint computation for a given set of facts and rules
- ✓ check programs regarding stratification
- ✓ name and explain two strategies for query evaluation

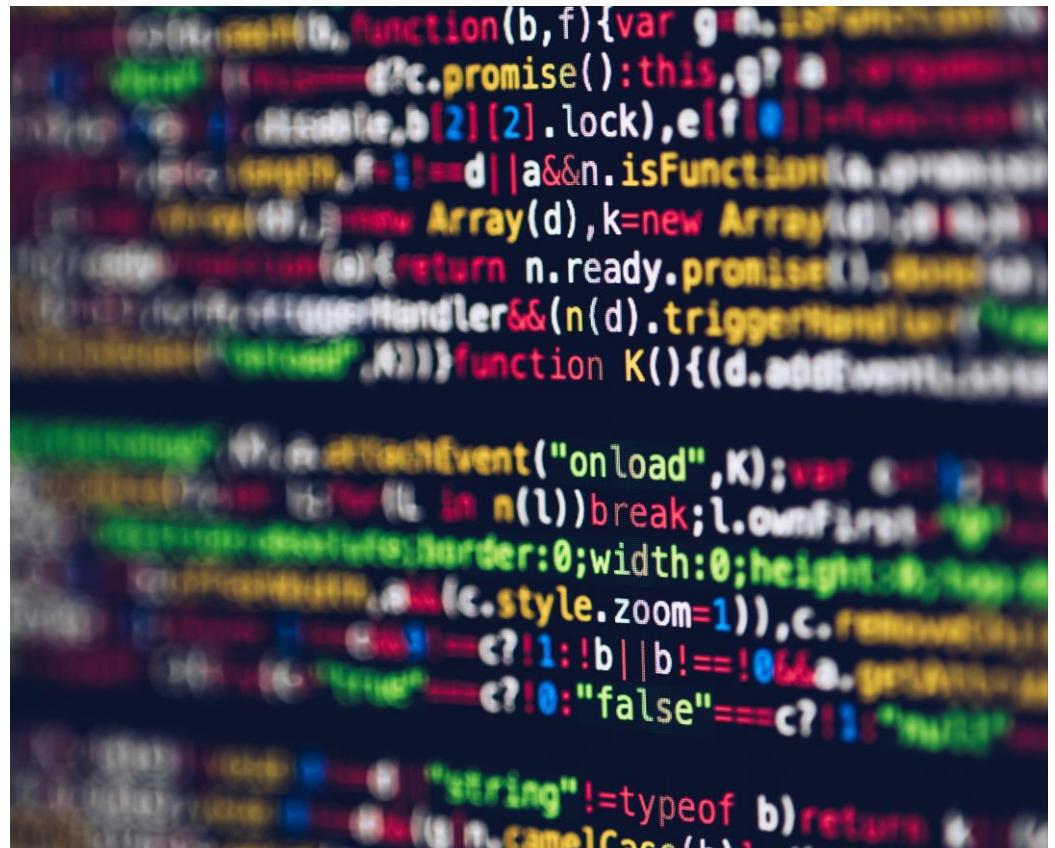


Motivation - Recursion

- Recursive queries and transitive closure cannot be expressed in SQL or relational algebra
- Example: How long does it take to fulfill task 89?

TaskID	SubtaskOf	Est. Time (h)	Responsible
145	123	8	Terry
123	100	16	Robert
100	89	32	Linda

- **Note:** Since SQL:99 Common Table Expressions (CTEs) for recursive queries, but not implemented in all systems



Motivation – Derive New Knowledge

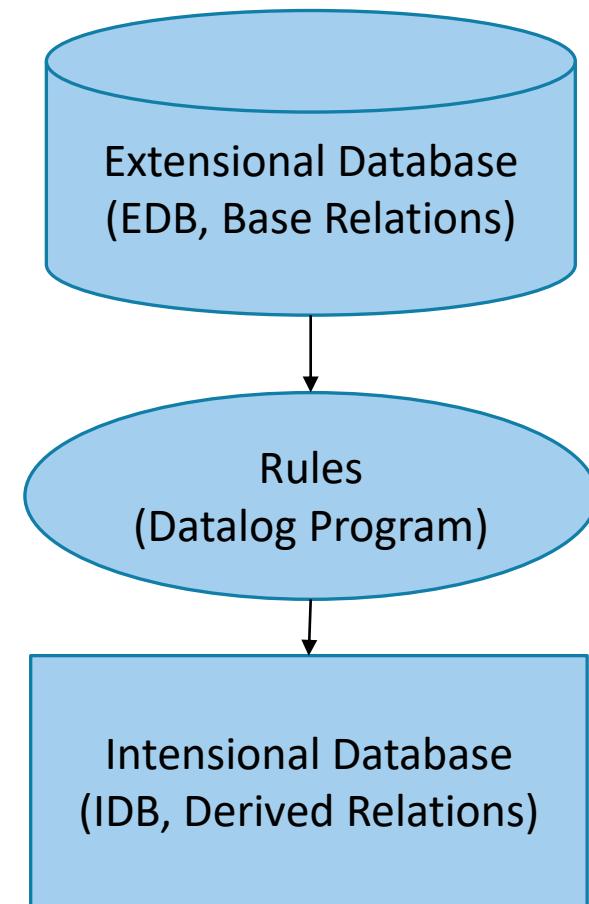
Related to items you've viewed



- Recommender Systems: you may be also interested in...
 - Derive new knowledge from the **facts** stored in your database using **rules**
 - Deductive Databases extend relational data model by a **deduction component**
- fuse data modeling with logic programming

Structure of Deductive Database Systems

- Extensional Database (or fact base)
 - set of relations and data
- Deduction Component with Datalog
 - Defines set of deduction rules
 - Datalog: rule language based on first order logic, subset of Prolog
- Intensional Database
 - Set of derived relations & data
 - Generated by executing Datalog rules in the deduction component



[Kemper & Eickler, 2015]

Recap: Domain Relational Calculus (see Chapter 2)

- Atomic formulas are $R(x_1, \dots, x_k)$ with R being a k-ary relation and x_1, \dots, x_k being variables or constants
- A database can be represented in the same way:
A database is a set of **facts**.
- A **fact** is an atomic relation predicate
with constants only representing one tuple of a relation → always true

DB Schema:

EMPL(eno, name, marstat, salary, dno)
DEPT(dno, dname, mgr)

Integrity constraints:

EMPL[dno] \subseteq DEPT[dno]
DEPT[mgr] \subseteq EMPL[eno]

```
empl(122,'Miller', 'single',35.000,4)
empl(101,'Meyer', 'married',55.000,4)
dept(4,'Anexa',101)
office(2,6232, 122)
...
```

Rules in Deductive DBs

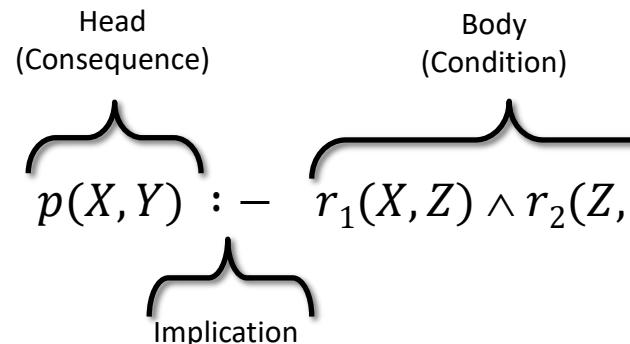
DB Schema:
EMPL(eno, name, marstat, salary, dno)
DEPT(dno, dname, mgr)
Integrity constraints:
EMPL[dno] ⊆ DEPT[dno]
DEPT[mgr] ⊆ EMPL[eno]

- Datalog rules are expressed as **Horn clauses** (in Conjunctive Normal Form)

$$p \quad :- \quad r_1 \wedge r_2 \wedge \dots \wedge r_k$$

where p and r_i are relation predicates.

- Example

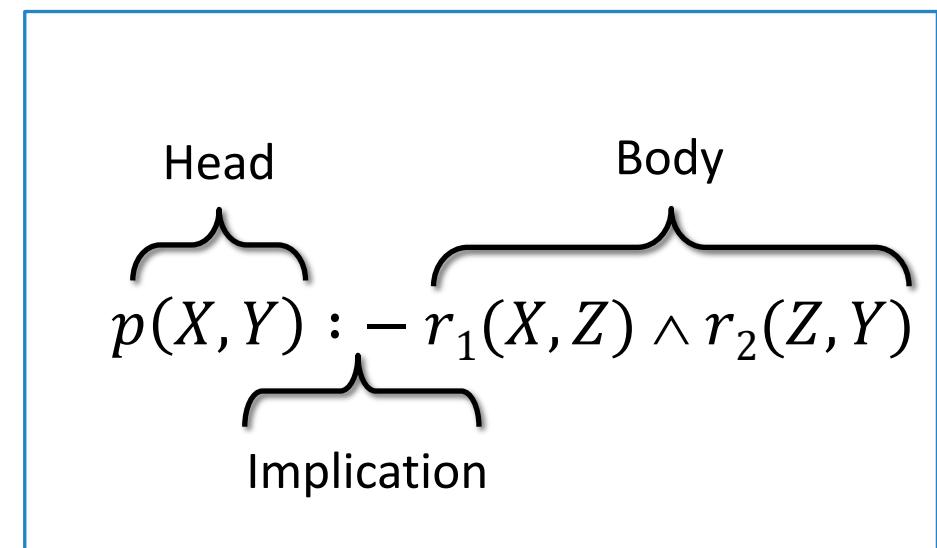


Semantics: $p(X, Y)$ is true, if there exist some X, Y, Z such that every r_i is true

manager(SSN, N) :- empl(SSN, N, M, S, D) \wedge dept(D, DN, SSN)

Syntax & Terminology

- Syntactic conventions
 - Predicate names start with a lower-case character
 - Variable names start with an upper-case character
 - String atoms are enclosed in ‘single quotes’
- Terminology
 - *Distinguished variables*: appear in head and body, e.g., X and Y (\forall)
 - *Non-distinguished (existential) variables*: appear only in body, e.g., Z (\exists)
 - *Anonymous variable*: $\underline{}$
- Datalog vs. Prolog
 - No predicates or functions as arguments
 - Only fix-point iteration
 - Efficient bottom-up evaluation



Queries and Constraints

- **Queries**

- *Goal clause* with "?-", e.g.,

$$? - \text{empl}(SSN, N, _, _, D) \wedge \text{dept}(D, 'Anexa', _)$$

- Usually, special *query predicate* is used, e.g.,

$$q(X, Y) : - r_1(X, Z) \wedge r_2(Z, Y)$$

- **Constraints:** Rules without head, that **must be always false**

$$: - \text{empl}(S, N, M, Salary, D) \wedge \text{Salary} < 10.000$$

More expressive constraints than
PKs and FKs are possible, similar
to Assertions in SQL

Examples

- Define rules for the Tax50 view from Chapter 3:

tax50 (SSN,N,M,S,D) :- empl (SSN,N,M,S,D) \wedge M='single' \wedge S < 40000

tax50 (SSN,N,M,S,D) :- empl (SSN,N,M,S,D) \wedge M='married' \wedge S < 80000

- Constraint: an employee must not earn more than his/her manager

:- empl (_,_,_,S,D) \wedge dept(D,_,M) \wedge empl (M,_,_,S2,_) \wedge S > S2

DB Schema:

EMPL(eno, name, salary, dno)

DEPT(dno, dname, mgr)

Integrity constraints:

EMPL[dno] \subseteq DEPT[dno]

DEPT[mgr] \subseteq EMPL[eno]

```
CREATE VIEW TAX50 AS
SELECT e.* FROM EMPL e
WHERE e.marstat='single' AND
e.salary<40.000) OR
(e.marstat='married' AND
e.salary<80.000);
```

Example

DB Schema:

EMPL(eno, name, salary, dno)
DEPT(dno, dname, mgr)

Integrity constraints:

EMPL[dno] \subseteq DEPT[dno]
DEPT[mgr] \subseteq EMPL[eno]

T: empl(12, 'jim', 50.000, 2).
 empl(11, 'jones', 60.000, 2).
 dept(2, 'R&D', 11).

works_dir_for(X,Y) :- empl(_,X,_,D), dept(D,_,M), empl(M,Y,_,_).

works_for(X,Y) :- works_dir_for(X,Y).
works_for(X,Y) :- works_dir_for(X,Z), works_for(Z,Y).

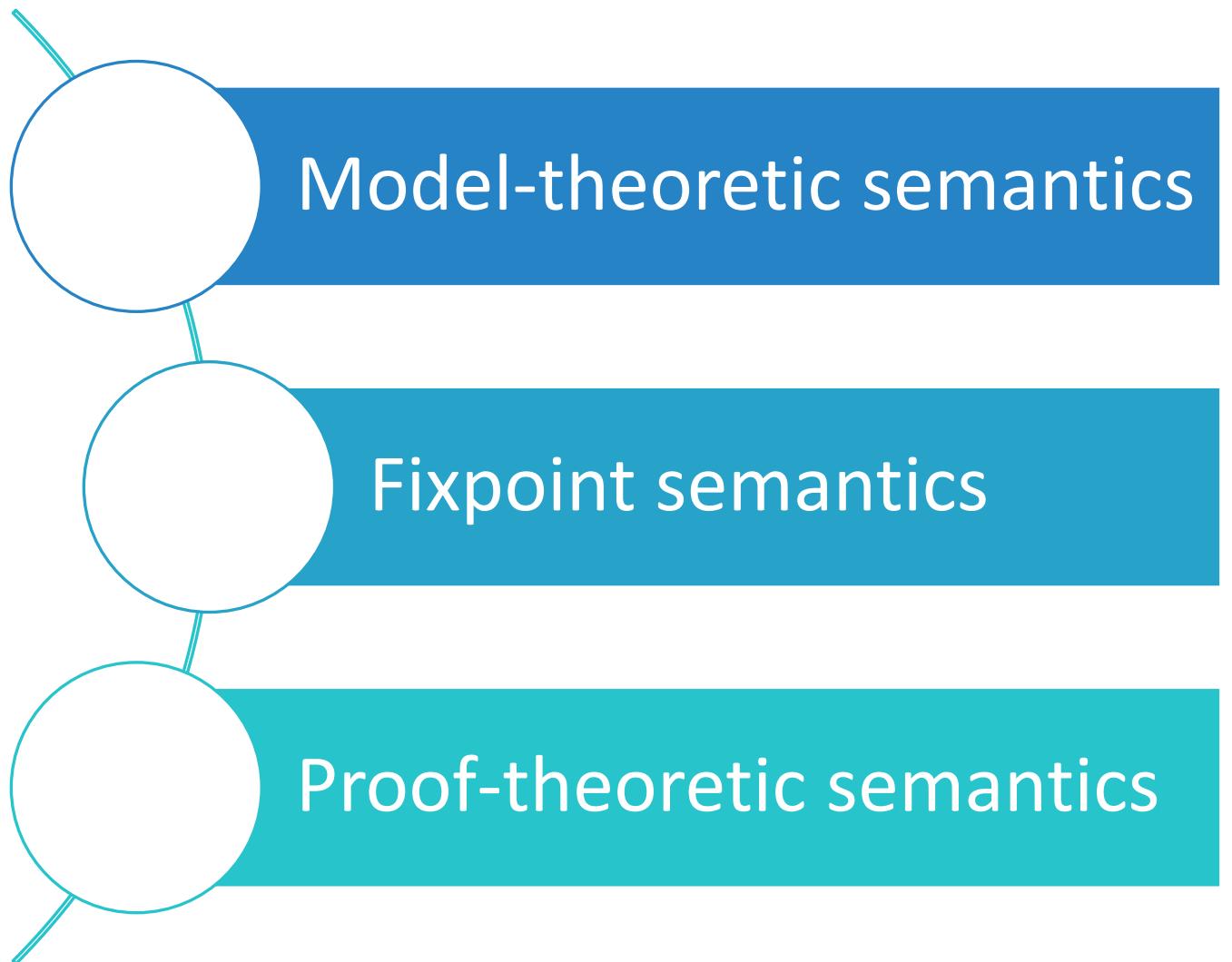
same_manager(X,Y) :- works_for(X,M), works_for(Y,M), X<>Y.

IC: :- empl(X,_,S,_), S<10.000 .
 :- empl(X,_,S,_), S>90.000 .

Query: ? - works_for(X, jones).

Approaches to Define Datalog Semantics

[Abiteboul et al., 1995]



Model-theoretic Approach

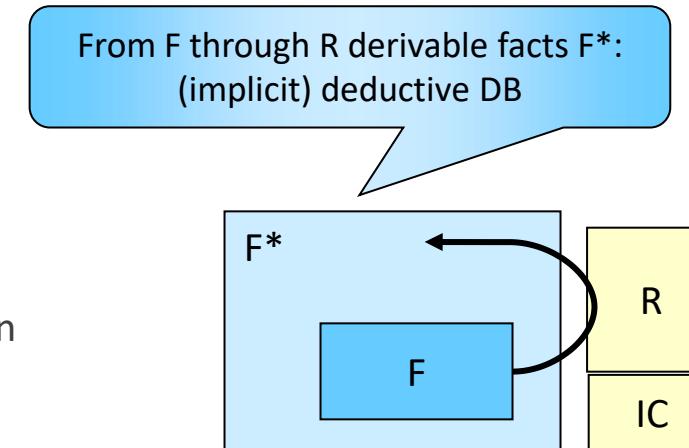
- Program P is a set of first-order rules (or first-order theory) describing the answer
- Model of P : database instance I that constitutes the result by satisfying the rules
- Semantics of P on input I : minimum model of P containing I
- Queries search through the instance
- Modifications must take the theory into consideration

Definition of DDBs

Definition 5.1

A deductive database consists of a set F of facts, a set R of deduction rules, a set IC of integrity constraints, and the set F^* of all explicit and derived (implicit) facts.

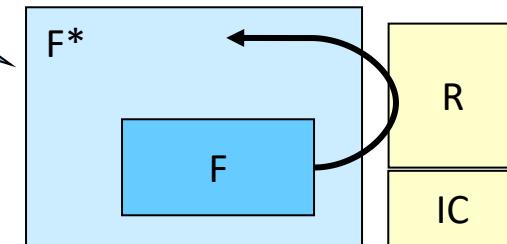
- Extensional DB (EDB)
 - Relations defined as a set of facts in F (*Base relations*)
- Intensional DB (IDB)
 - Relations defined by rules in R (*Derived relations*)
- Datalog program
 - Query acting on **extensional databases** and producing a result that is an **intentional database** → mapping EDBs to IDBs
- Datalog and its variations:
 - **Datalog \neg** (Negation is allowed),
 - **NR-Datalog** (Recursion is not allowed)



Semantics of Deductive Databases

- Deductive database $D=(F, R, IC)$
 - Questions
 - What is the formal definition of F^* ?
 - Is F^* uniquely determinable?
 - What is the meaning of “derivable”?

F*: implicit relational database represented by D



- Problems are caused by negation and recursion → classes NR-DATALOG, NR-DATALOG \neg and DATALOG \neg have to be considered separately

Semantics of DDBs: NR-Datalog

F^* is formally defined as the minimal *Herbrand model* of D.

Definition 5.2 (Herbrand Base of D):

All positive **ground literals** that are constructable from **predicates** in D and **constants** in D.

Definition 5.3 (Herbrand Model of D):

A *Herbrand Model* is every subset M of the Herbrand Base of D, such that:

Every fact from F is contained in M. For every **ground instance** of a rule in D over constants in D, if M contains **all literals** in the **body**, then M contains the **head as well**.

A minimal model does not properly contain any other model.

Example: NR-DATALOG

F: $q(a,b)$ $t(b)$

$q(b,a)$

R: $p(X) \leftarrow q(X,Y), t(Y)$

Herbrand Base

$q(a,a)$	$q(a,b)$	$p(a)$	$t(a)$
$q(b,a)$	$q(b,b)$	$p(b)$	$t(b)$

Minimal

Herbrand Models M_i

$M_1:$	$q(a,b)$	$t(b)$
	$q(b,a)$	$p(a)$

$M_2:$	$q(a,b)$	$t(b)$	$p(a)$
	$q(b,a)$		$p(b)$
	$q(b,b)$		

The ground instances of the rule are thus (blue = contained in M_i)

$p(a) \leftarrow q(a,a), t(a)$
$p(a) \leftarrow q(a,b), t(b)$
$p(b) \leftarrow q(b,a), t(a)$
$p(b) \leftarrow q(b,b), t(b)$

$p(a) \leftarrow q(a,a), t(a)$
$p(a) \leftarrow q(a,b), t(b)$
$p(b) \leftarrow q(b,a), t(a)$
$p(b) \leftarrow q(b,b), t(b)$

Least Fixpoint for NR-Datalog

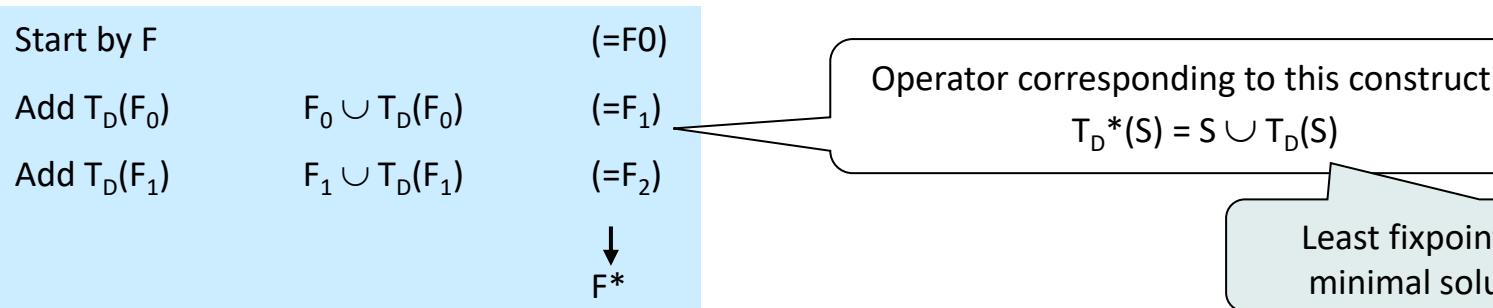
F^* is created by the repeated (finite) application of the immediate consequence operator T_D (naive evaluation strategies) starting from F results in derivation of implicit facts from F :

$$T_D(T_D(\dots(T_D(F))\dots))$$

Definition 5.4:

For a subset S of a Herbrand Base the application of T_D to S is defined as:

$$T_D(S) := \{ H : (H \leftarrow B) \text{ is a ground instance of a rule with head } H \text{ and body } B \text{ such that } S \text{ contains all literals in } B \}$$

**Theorem 5.1:** (v. Emden, Kowalski 1976)

The uniquely determined least fix point of T_D^* is the minimal Herbrand Model of D

Example: Least Fixpoint Computation

$q(X) :- r(X,Y), \text{NOT } b(X).$

$r(X,Y) :- c(X,Y), b(Y).$

$r(X,Y) :- c(X,Z), r(Z,Y).$

$c(1,2)$ $b(3)$

$c(2,3)$ $b(4)$

$c(1,4)$

Semantics of DDBs: NR-DATALOG

Problem: If negation occurs in the body of a rule, then there may be no unique minimal Herbrand model anymore.

Which one is the “natural model” of D, and thus represents the intended semantics of D?

Example:

$$\begin{array}{ll} F: & q(a,b) \quad t(b) \\ & q(b,a) \end{array}$$

$$R: p(X) \leftarrow q(X,Y), \text{NOT } t(Y)$$

- The minimal Herbrand models are
- Ground instances of the rules (in M_1 , in M_2 , in M_1 and M_2):

$$\begin{array}{lll} p(a) & \leftarrow & q(a,a), \text{NOT } t(a) \\ p(a) & \leftarrow & q(a,b), \text{NOT } t(b) \\ p(b) & \leftarrow & q(b,a), \text{NOT } t(a) \\ p(b) & \leftarrow & q(b,b), \text{NOT } t(b) \end{array}$$

$$\begin{array}{ll} M_1: & F \cup \{p(b)\} \\ M_2: & F \cup \{t(a)\} \end{array}$$

p(b) is derivable from F, M_1 is the
“natural” model of D

No reason why t(a) should be true.

- Main problem in presence of negation: characterization of the “natural” model of D (representing the intended semantics of D).

Example (NR-DATALOG \neg)

Least fixpoint characterization cannot be directly adopted

$$\begin{array}{ll} F: & q(a,b) \quad s(b) \\ & q(b,a) \end{array}$$

$$\begin{array}{lll} R: & p(X) & \leftarrow q(X,Y), \text{NOT } t(Y) \\ & t(Y) & \leftarrow s(Y) \end{array}$$

First application of T_D :

- No t-fact in $F \Rightarrow p(a)$ and $p(b)$ are derivable
- $s(b)$ in $F \Rightarrow t(b)$ is derivable

Second application of T_D :

- No new fact derivable \Rightarrow fixpoint reached

It follows therefore:

- least fixpoint of T_D^* : “natural” Herbrand model: $F \cup \{t(b), p(a), p(b)\} \neq F \cup \{t(b), p(b)\}$

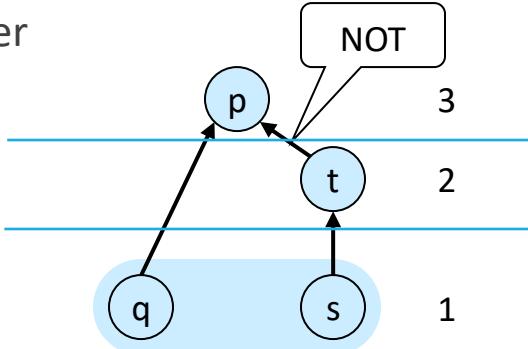
Reason (intuitively): “ $t(b)$ arrives too late for preventing derivation of $p(a)$.”

F: $q(a,b) \quad s(b)$
 $q(b,a)$

R: $p(X) \leftarrow q(X,Y), \text{NOT } t(Y)$
 $t(Y) \leftarrow s(Y)$

Stratification

- Predicates “call” each other in a hierarchical order



- Predicates can be *layered* (or: *stratified*); no two predicates in a layer (*stratum*) depend negatively on each other. If a predicate p depends on a negative predicate r , then r is in a lower layer.
- If application of T_D is done layer by layer, the least fixpoint of D is consequently the *natural* Herbrand model.
- It follows therefore:
 1. layer: F
 2. layer: $F \cup \{t(b)\}$
 3. layer: $F \cup \{t(b)\} \cup \{p(b)\}$

Least Fixpoint for Recursive Stratified DATALOG \neg Programs

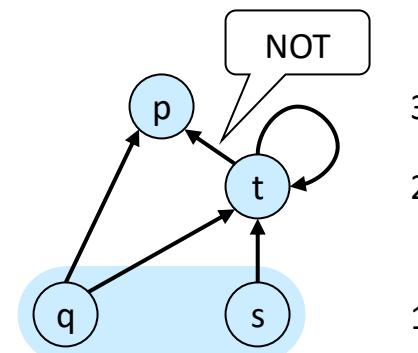
- **Problem:** The recursion leads possibly to more than one application of T_D per layer.
If negations do not occur in the recursion cycle, there will be no problem.

Example:

F:	$q(b,c)$	$s(c)$	R:	$p(X) \leftarrow q(X,Y), \text{NOT } t(Y)$
				$t(Y) \leftarrow q(Y,Z), t(Z)$
				$t(Y) \leftarrow s(Y)$

From this it follows:

1. layer: F
2. layer: $F \cup \{t(c)\}$
 $F \cup \{t(c)\} \cup \{t(b)\}$
3. layer: $F \cup \{t(c)\} \cup \{t(b)\} \cup \{p(b)\}$
.....



Non-stratified DATALOG \neg Programs

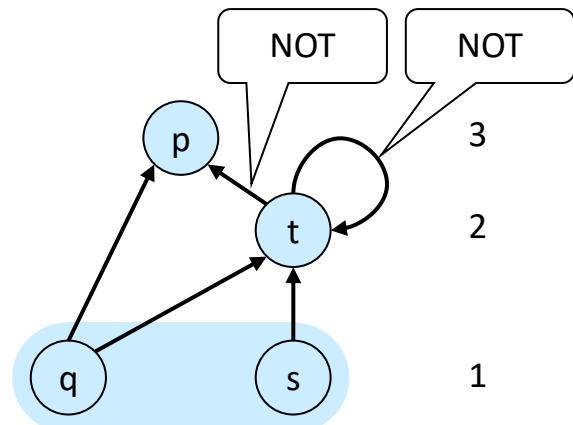
- **Problem:** Negation in the recursion cycle violates the stratification condition.

Example:

$$\begin{array}{ll} F: & q(a,b) \quad s(c) \\ & q(b,a) \\ R: & p(X) \leftarrow q(X,Y), \text{NOT } t(Y) \\ & t(Y) \leftarrow q(Y,Z), \text{NOT } t(Z) \\ & t(Y) \leftarrow s(Y) \end{array}$$

Even when applying T_D for t-rules only results in „anomalies“:

- $t(a)$ is generated, as $t(b)$ is initially missing
- $t(b)$ is generated, as $t(a)$ is initially missing
- Do $t(a)$ and $t(b)$ belong to the “natural” Herbrand model now?
Is there any reasonable “natural” Herbrand model at all?



F:	$q(a,b)$	$s(c)$	R:	$p(X) \leftarrow q(X,Y), \text{NOT } t(Y)$
	$q(b,a)$			$t(Y) \leftarrow q(Y,Z), \text{NOT } t(Z)$
				$t(Y) \leftarrow s(Y)$

Example (cont.)

- $t(a)$ and $t(b)$ are not “natural” consequences of $R \cup F$:

$t(a) \leftarrow q(a,Z), \text{NOT } t(Z)$
 b b

$t(b) \leftarrow q(b,Z), \text{NOT } t(Z)$
 a a

- Consequently, it results in two exclusive cases:

either $\text{NOT } t(b)$, consequently $t(a)$
or $\text{NOT } t(a)$, consequently $t(b)$.

- So that it yields also two different models:

$F \cup \{p(a), t(a)\}$, also $\text{NOT } t(b)$
or $F \cup \{p(b), t(b)\}$, also $\text{NOT } t(a)$.

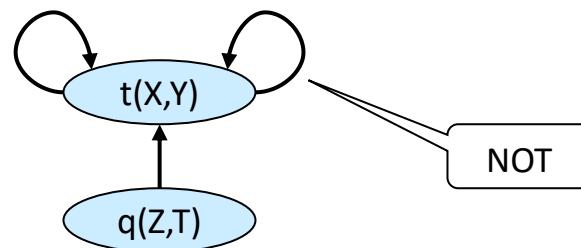
Local Stratification

Example DATALOG[¬] [cont.]:

$$\begin{array}{ll} F: & q(a,b) \\ & q(b,c) \\ R: & t(X,Y) \leftarrow q(X,Y), \text{NOT } t(Y,X) \\ & t(X,Y) \leftarrow t(X,Z), t(Z,Y), \text{NOT } t(Y,X) \end{array}$$

R is not stratified, but “locally stratified”:

“Reachable” instances of R:



$$\begin{aligned} t(a,b) &\leftarrow q(a,b), \text{NOT } t(b,a) \\ t(b,c) &\leftarrow q(b,c), \text{NOT } t(c,b) \\ t(a,c) &\leftarrow t(a,b), t(b,c), \text{NOT } t(c,a) \end{aligned}$$

- No recursion cycle involves negation among “reachable” instances.
- Test for local stratification expensive, but can be computed “on the fly”, i.e., during evaluation of the rules without computing the (local) stratification [Bry, 1989]

Are these programs stratified?

Program 1

$q(X) :- \text{NOT } p(X), t(X).$

$p(X) :- s(X,X), \text{NOT } r(X).$

$s(X,Y) :- s(Y,X), t(Y).$

$r(X) :- t(X), \text{NOT } s(X,X).$

Program 2

$p(X) :- q(X,Y), \text{NOT } t(Y).$

$t(Y) :- q(Y,Z), \text{NOT } t(Z).$

$t(Y) :- s(Y).$

Proof-theoretic / Axiomatic Approach

- Theory: Facts and deduction rules (T) + integrity constraints (IC)
- Queries are theories, which must be proved by using the axioms T .
- A modification adopts a new theory (e.g., T') and it should not violate the integrity constraints, i.e.,

$$T' \models IC$$



T' is a model for IC or IC is valid under T'

Bottom-Up vs. Top-Down Evaluation

Bottom-Up (Forward Chaining)

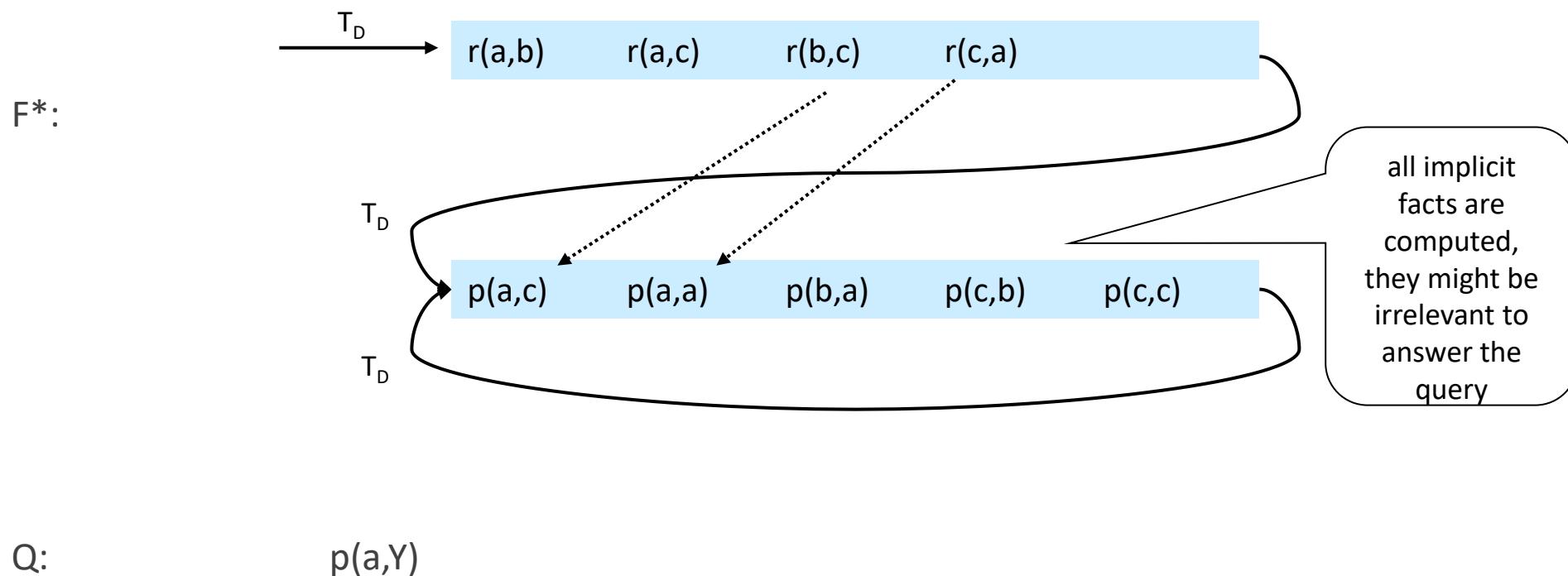
- Generation of implicit facts at evaluation-time
- Evaluation of the query against temporarily materialized implicit databases.
(direct implementation of least fixpoint computation)
- **Drawback:** when materializing F^* , the particular query Q is not considered → many irrelevant answers and intermediate results may be generated.

Top-Down (Backward Evaluation)

- Generation of subqueries until queries to base relations are reached
- Evaluation of base subqueries against F and upward propagation of answers to the top query
- As opposed to bottom-up approach:
constants in top query and subqueries are passed downwards and provide restrictions while evaluating base queries.
- **Drawback:** Inefficient (or not terminating) for recursive queries

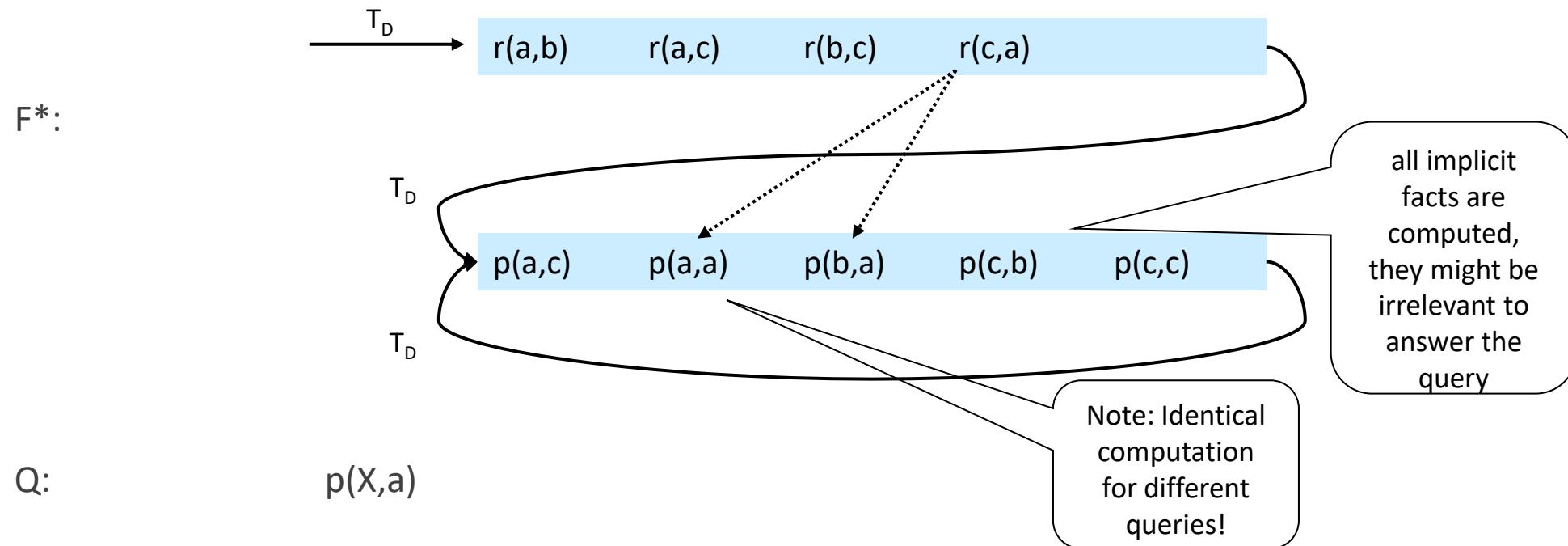
F:	$q(a,b)$	$q(a,c)$
	$q(b,c)$	$q(b,d)$
	$q(c,a)$	$t(d)$
R:	$p(X,Y) \leftarrow q(X,Z), r(Z,Y)$	
	$r(Z,Y) \leftarrow q(Z,Y), \text{NOT } t(Y)$	

Example: Bottom-Up Evaluation



Example: Bottom-Up Evaluation

F:	q(a,b)	q(a,c)
	q(b,c)	q(b,d)
	q(c,a)	t(d)
R:	$p(X,Y) \leftarrow q(X,Z), r(Z,Y)$	
	$r(Z,Y) \leftarrow q(Z,Y), \text{NOT } t(Y)$	



F:	$q(a,b)$	$q(a,c)$
	$q(b,c)$	$q(b,d)$
	$q(c,a)$	$t(d)$
R:	$p(X,Y) \leftarrow q(X,Z), r(Z,Y)$	
	$r(Z,Y) \leftarrow q(Z,Y), \text{NOT } t(Y)$	

Example: Top-Down Evaluation

Query: $p(a,Y)$

- Match query with head of 1st rule and generate subqueries for body → $q(a,Z), r(Z,Y)$
- *Choice Point* → $q(a,Z)$ can be proven with facts from F, e.g., take $q(a,c)$ → $q(a,c), r(c,Y)$
- Prove $r(c,Y)$ by using second rule → $q(c,Y), \text{NOT } t(Y)$
- $q(c,Y)$ can be proven by $q(c,a)$ in F → $q(c,a), \text{NOT } t(a)$
- NOT $t(a)$ can be also proven by F, thus, $r(c,a)$ and $p(a,c)$ are true
- Derivation of one result finished → $p(a,c)$
- Backtrack to choice point and generate next result (if required)

Integrity Constraints

Definition 5.5:

Integrity constraints (IC) are conditions that have to be satisfied by a database at any point in time (expressing general laws which cannot be used as derivation rules).

Definition 5.6:

Integrity-checking tests, whether a particular update is going to violate any constraint.

- **Main problem for IC-tests:**
Full evaluation of all ICs before every update would be very expensive and would decrease update performance significantly.
- **Solution:**
Determine a reduced set of **simplified ICs** for which the checking guarantees satisfaction of **all ICs**.
- This approach leads to a **specialization of constraints**.

Representation of Constraints and Updates

- Integrity constraints expressed as **Datalog**[¬] rules for a meta-predicate “inconsistent”.

inconsistent \leftarrow employee(X), works_for(X,X) (original constraint: no employee works for himself)

- Updates are either **atomic** insertions / deletions of facts or **general** updates
(depending on a condition, which is a Datalog query to be evaluated before the update)

insert works_for(john, jim)

delete employee(X) where works_for(X, john)

Constraint Specialization

Input:

- Update $U=\{\text{delete } L, \text{ insert } L\}$
- Integrity constraints IC (satisfied before the update)

1. IC is affected by U, if IC contains a literal L^* that is unifiable with L (resp. NOT L), if U is an insertion (resp. deletion).
2. For every such L^* , IC_{σ} is a relevant instance of IC with respect to U. (where σ is a most-general-unifier of L and L^*)
3. Simplified relevant instances of IC with respect to U are obtained by deleting L^* from relevant instances.

So far, generalized updates, derivation rules, and negations have not been taken into account yet.

Example: Constraint Specialization

IC: inconsistent $\leftarrow p(X,Y), \text{NOT } s(X)$ (corresponding: FORALL X,Y: $p(X,Y) \Rightarrow s(X)$)

- insert $p(a,b)$
inconsistent $\leftarrow p(a,b), \text{NOT } s(a)$ IC affected!
- delete $p(a,b)$
- insert $s(a)$
- delete $s(a)$
inconsistent $\leftarrow p(a,Y), \text{NOT } s(a)$ IC affected!

Generalized Updates

Example:

IC: inconsistent $\leftarrow p(X,Y), \text{NOT } s(X)$

U: Insert $p(X,b)$ where $r(X)$

1. IC is affected by U, if IC contains a literal L^* that is unifiable with L (resp. NOT L), if U is an insertion (resp. deletion).
2. For every such L^* IC_{σ} is a relevant instance of IC regarding U.
3. Simplified relevant instances of IC regarding U are obtained by deleting L^* from relevant instances.

Instead of a simplified relevant instance, we end up with a specialized constraint:

inconsistent $\leftarrow r(X), \text{NOT } s(X)$



5.3 Queries in Data Integration Systems

Motivation

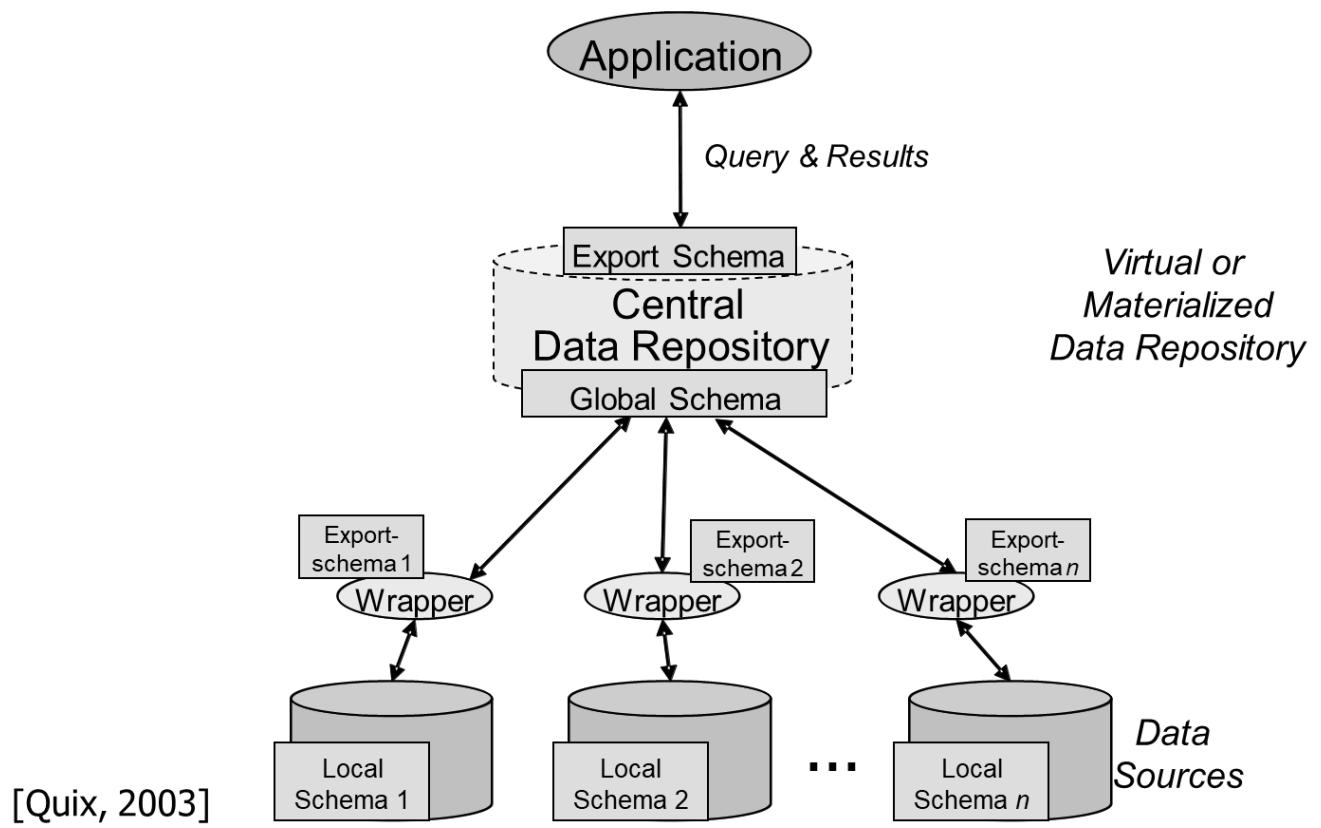
Definition 5.7:

Data integration is the problem of providing unified and transparent access to a collection of data stored in multiple, autonomous, and heterogeneous data sources

Motivations for data integration

- Company mergers
- Reorganization of companies
- Restructuring of databases in an organization
- Combination of data from several internal and external sources for data analysis (e.g. data warehousing, OLAP)

General Integration Architecture



Data Integration Approaches

- Distributed databases
 - Data sources are homogeneous DBs under central control
- Multi-database or federated databases
 - Data sources are autonomous, heterogeneous databases
- **(Mediator-based) Data Integration**
 - Access through a global schema mapped to autonomous and heterogeneous data sources
- Data Exchange
 - Mapping between a source and a target system
 - Source may specify data in target only partially
- Peer-to-peer data integration
 - Network of autonomous systems mapped one to each other, without a global schema

Challenges

- Modeling of the global schema, the sources, and **the mappings between the two**
 - Construction of the global schema
 - Discovering mappings between sources and global schema
- Legacy databases: DBs are used for many applications, structure cannot be changed.
- Heterogeneity & Conflicts
 - Data Types & Structures
 - Semantics: Meaning of terms on schema- and instance-level
- **Answering queries expressed on the global schema**
- Data extraction, cleaning, and reconciliation
- Processing of updates expressed on the global schema a/or the sources (“read/write” vs. “read-only” data integration)
- Optimization of queries across data sources

Querying Problem

- A query expressed in terms of the global schema must be reformulated in terms of (a set of) queries over the sources and/or materialized views
- The computed sub-queries are shipped to the sources, and the results are collected and assembled into the final answer
- The computed query plan should guarantee
 - completeness of the obtained answers wrt the semantics
 - efficiency of the whole query answering process
 - efficiency in accessing sources

Mappings

How to specify the mapping between the data sources and the global schema?

- **LAV (local-as-view, source-centric):**

The sources are defined in terms of the global schema (i.e., as view on the global schema)

- **GAV (global-as-view, global-schema-centric):**

The global schema is defined in terms of the sources (i.e., as view on the source schemas)

- **GLAV (combination of GAV and LAV)**

Example

- Source Schema S
 - $em50(\text{Title}, \text{Year}, \text{Director})$
European movies since 1950
 - $rv10(\text{Movie}, \text{Review})$
reviews since 2000
- Global Schema G
 - $\text{movie}(\text{Title}, \text{Director}, \text{Year})$
 - $\text{ed}(\text{Name}, \text{Country}, \text{Dob})$ (*European directors*)
 - $\text{rv}(\text{Movie}, \text{Review})$
- Query q

```
SELECT M.title, R.review
FROM Movie M, RV R
WHERE M.title=R.title AND M.year = 2000
```



Questions



- Can we rewrite q as a query over the source schema S ?
- Is there a unique solution?
- If not, can we characterize a best solution?
- What is the semantics of a query?
- And how do we specify the mapping between S and G ?

Problem: Expressivity

- In order to answer such questions, we have to prove that queries are equivalent or contained in each other

- Query Equivalence
 - q and q' are equivalent if they produce the same result for all legal databases
- Query Containment
 - q is contained in q' if the result of q is a subset of the result for q' for all legal databases
- Undecidable for many query languages
- Restriction to conjunctive queries → Tableau Method

GAV Example



- movie>Title, Year, Director)
:- em50>Title, Year, Director).

- ed>Name)
:- em50_,_,Name).

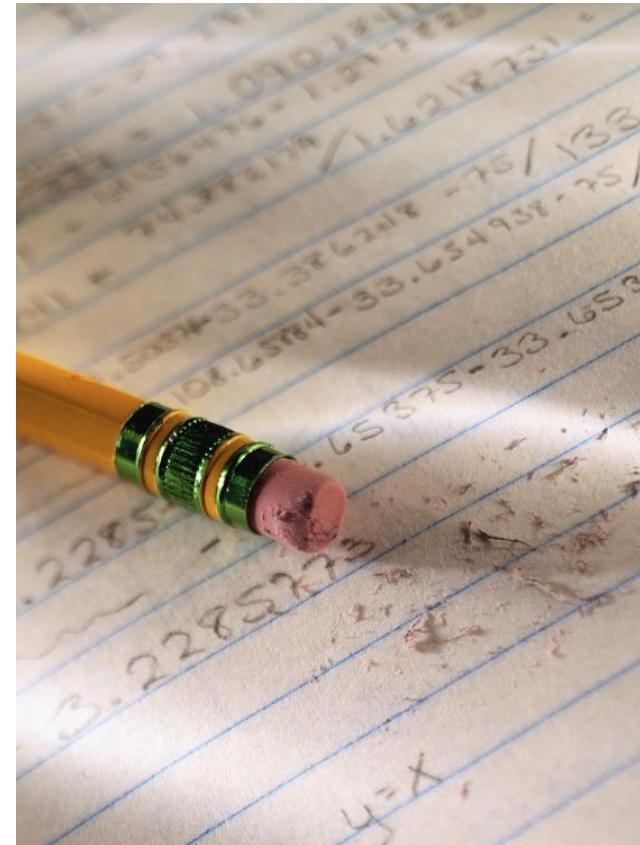
- rv>Movie, Review)
:- rv10>Movie, Review).

Query Rewriting in GAV

Queries over G can be rewritten as queries over S by unfolding

```
q>Title, Review) :-  
    movie>Title, 2000, _),  
    rv>Title, Review).
```

```
q'(Title, Review) :-  
    em50>Title, 2000, _),  
    rv10>Title, Review).
```



LAV Example

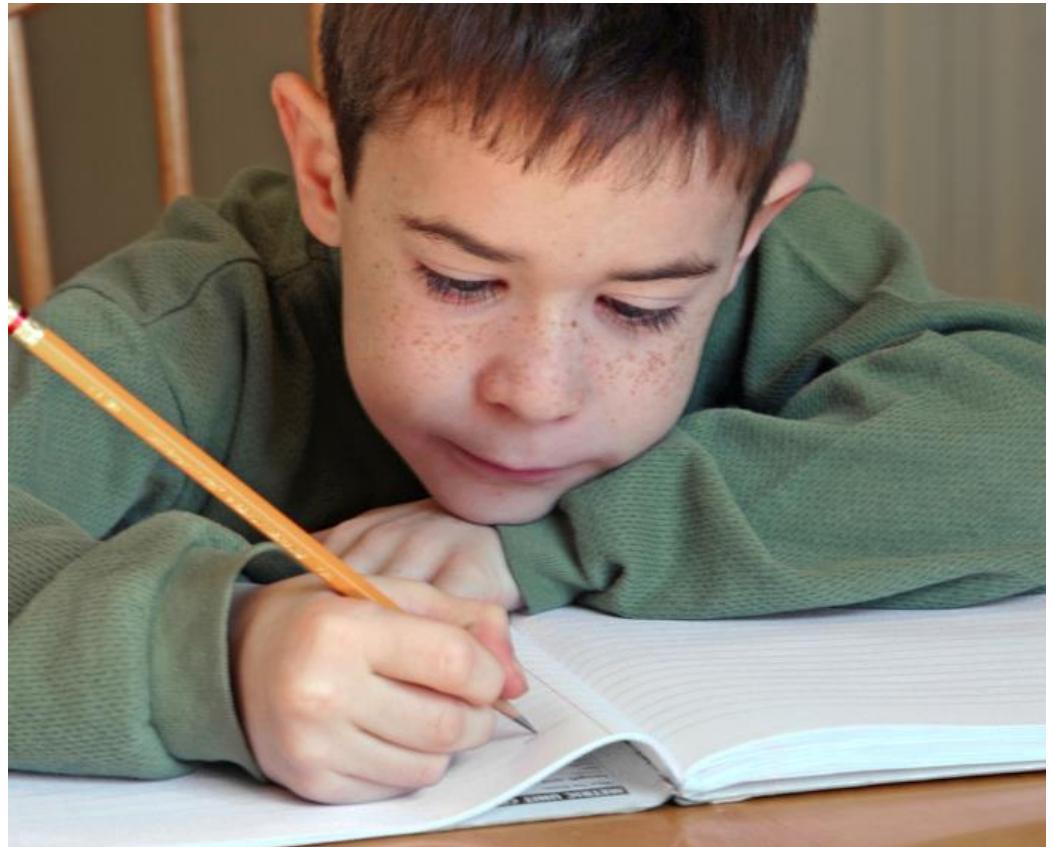


- em50>Title,Year,Director) :-
movie>Title,Year,Director),
ed(Director,Country,Dob),
Year \geq 1950.

- rv10>Movie,Review) :-
rv>Movie,Review),
movie>Movie,Year,Director),
Year \geq 2000.

Query Rewriting in LAV

- Sources are views
 - Answer queries on the basis of available data in the views
 - Research area: Answering queries using views
- **Query Optimization:** Answer a query using a materialized view!
- **Idea:** Try to cover the predicates of the query by predicates in the bodies of the source views



Query Rewriting in LAV Example

- `q>Title, Review) :-
movie>Title, Year, _),
rv>Title, Review),
Year = 2000.`

- `em50>Title, Year, Director) :-
movie>Title, Year, Director),
ed(Director, Country, Dob),
Year ≥ 1950.`

- `rv10(Movie, Review) :-
rv(Movie, Review),
movie(Movie, Year, Director),
Year ≥ 2000.`

Summary

- **Query Processing in Big Data**

- Data partitions & replications enable parallel, distributed, fault-tolerant query processing
- MapReduce is a generic programming pattern for distributed computation, but often too rigid for complex data analysis
- High-Level languages enable declarative query processing in distributed systems

- **Deductive databases are the logical basis for relational databases**

- Herbrand model corresponds to least fixpoint of T_D
- Negation and recursion require stratified computation of fixpoints
- Integrity constraints are a special form of rules

- **Application in Information Integration**

- Mappings between data sources and integrated schema can be represented as logical rules
- Queries to integrated schema have to be rewritten into queries for data sources
- Mappings can be specified as LAV or GAV mappings

Review Questions

- Explain the Map-Reduce programming pattern? What is done in the Map function, what is done in Reduce?
- What are the problems of the Map-Reduce programming pattern?
- What is the goal of systems like Pig Latin or Hive?
- What is a broadcast hash join in Spark?
- When do you need to do shuffling in Spark? What are narrow and wide dependencies?
- Sketch and explain the data warehouse architecture!
- What is the Herbrand Base and the Herbrand Model?
- How can you compute the minimal Herbrand Model?
- Translate a RA query into Datalog and vice versa!
- What is stratification?
- Explain top-down and bottom-up evaluation of Datalog programs!
- Show an example for a mapping between a source schema and an integrated schema using the Datalog notation.
- What is the difference of GAV and LAV mappings?

References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., & Vianu, V. (1995). Foundations of databases (Vol. 8). Reading: Addison-Wesley. <https://wiki.epfl.ch/provenance2011/documents/foundations+of+databases-abiteboul-1995.pdf>

[Albrecht, 2018] Jens Albrecht. Big Data-Technologien – Überblick, <http://docplayer.org/77320835-Big-data-technologien-qual-der-wahl-prof-dr-jens-albrecht.html>(2018)

[Bry, 1989] Bry, F. Logic programming as constructivism: A formalization and its application to databases Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1989, 34-50

[Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). Fundamentals of database systems (Vol. 7). Pearson

[Gottlob et al, 2012] Ceri, S.; Gottlob, G. & Tanca, L. Logic programming and databases Springer Publishing Company, Incorporated, 2012

[Karau & Warren, 2017] Karau, H., & Warren, R. (2017). *High performance Spark: best practices for scaling and optimizing Apache Spark*. O'Reilly Media, Inc.

[Kemper & Eickler, 2015] Kemper, A., & Eickler, A. (2015). *Datenbanksysteme*. Oldenbourg Wissenschaftsverlag

[Olston et al., 2008] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110

[Quix, 2003] Quix, C. Metadata Management for quality-oriented Information Logistics in Data Warehouse Systems (in German) RWTH Aachen University, 2003

[Schöning, 2000] Schöning, U. Logik für Informatiker Spektrum Akademischer Verlag, 2000

[Thusoo et al., 2009] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghatham Murthy: Hive - A Warehousing Solution Over a Map-Reduce Framework. PVLDB 2(2): 1626-1629 (2009)

Implementation of Databases

Chapter 6: Advanced Transaction Management

Winter Term 23/24

Lecture

Prof. Dr. Sandra Geisler

Excercises

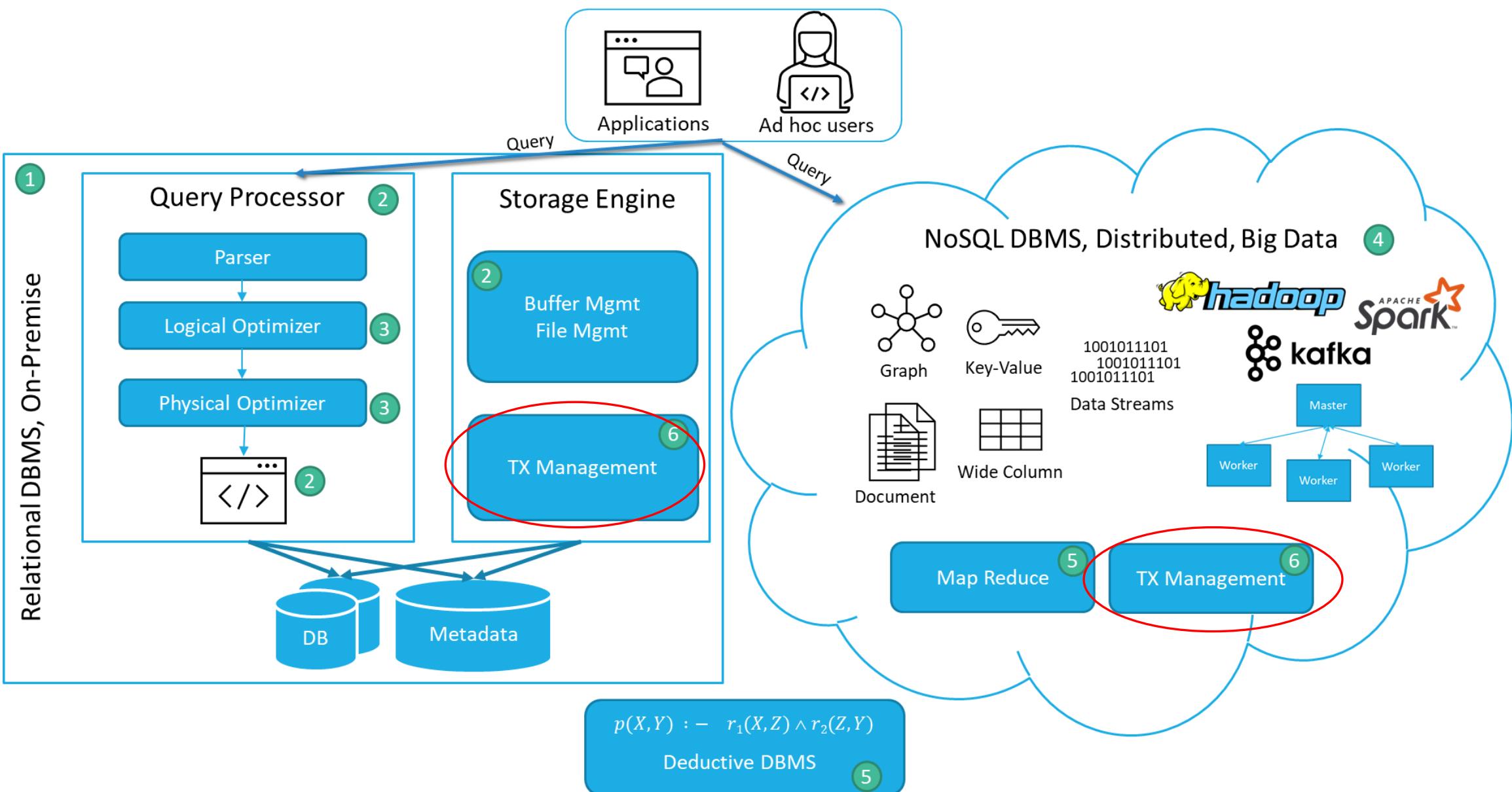
Anastasiia Belova, M.Sc.

Soo-Yon Kim, M.Sc.



Juniorprofessur
für Datenstrom-
Management
und -Analyse

RWTHAACHEN
UNIVERSITY



Overview

6.1 Synchronization Problems

6.2 Read-Write-Model

6.3 Serializability

6.4 Transaction Recovery

6.5 Concurrency Control Protocols

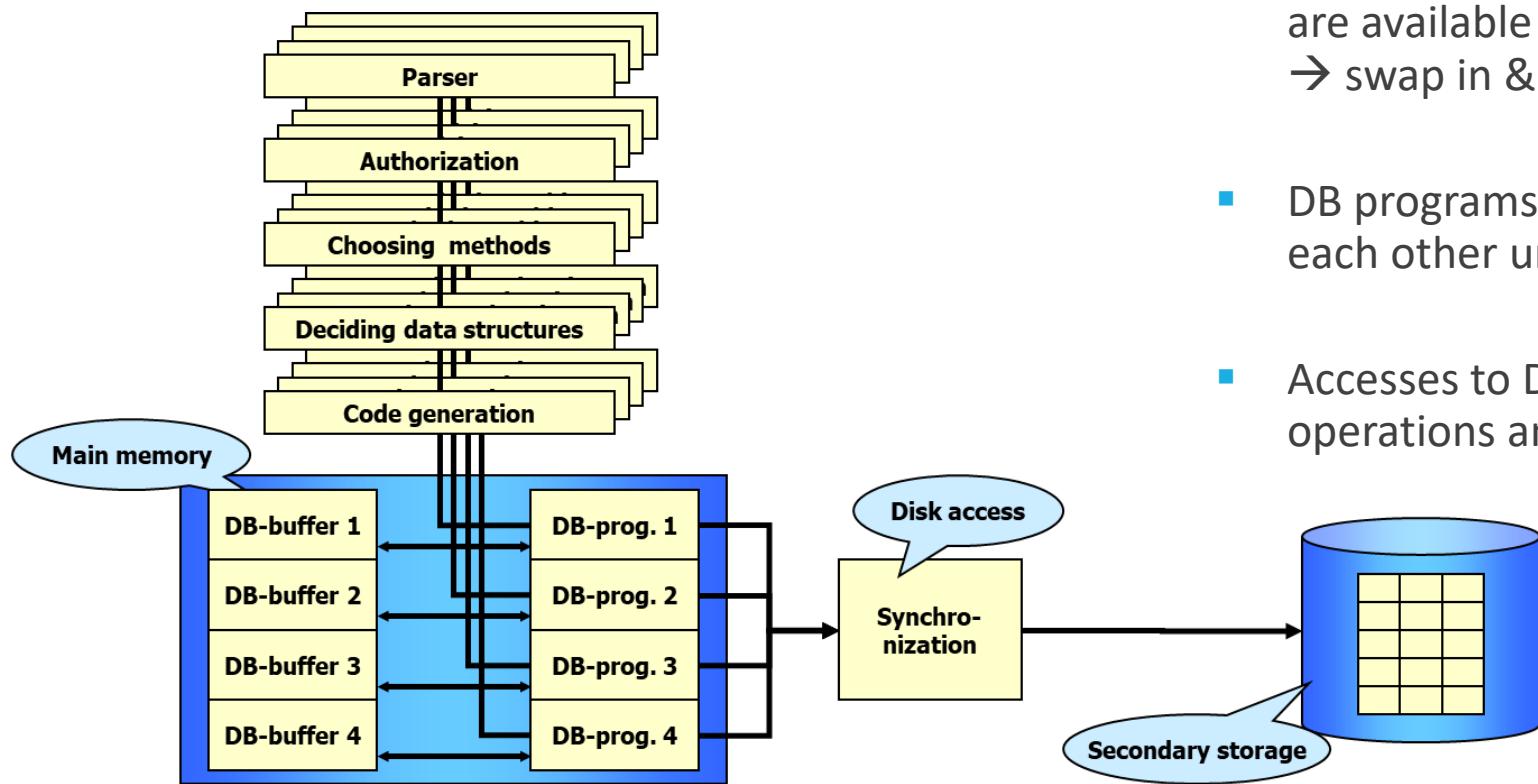
6.6 Index Locking

6.7 Recovery Protocols

6.8 Concurrency Control in SQL

6.9 Distributed Transactions

Synchronization Layers in DB Systems



- During the execution, part of DB buffers are available for a DB program
→ swap in & out processed data
- DB programs work independently from each other under control of OS
- Accesses to DBs result from read/write operations and must be synchronized

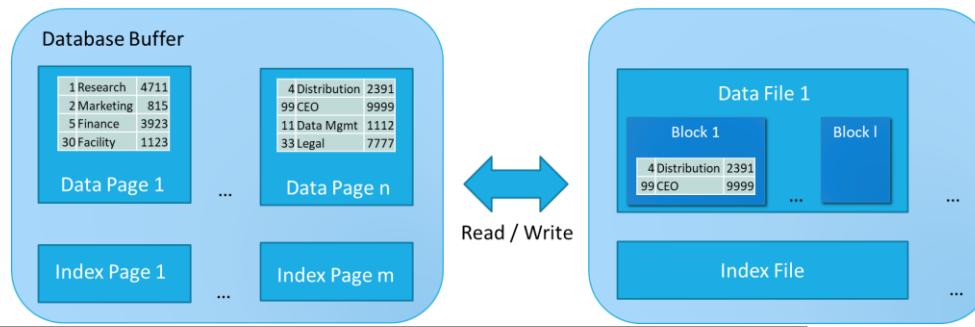
Locking Layers

Comparison of locking layers (transactions T1, T2)

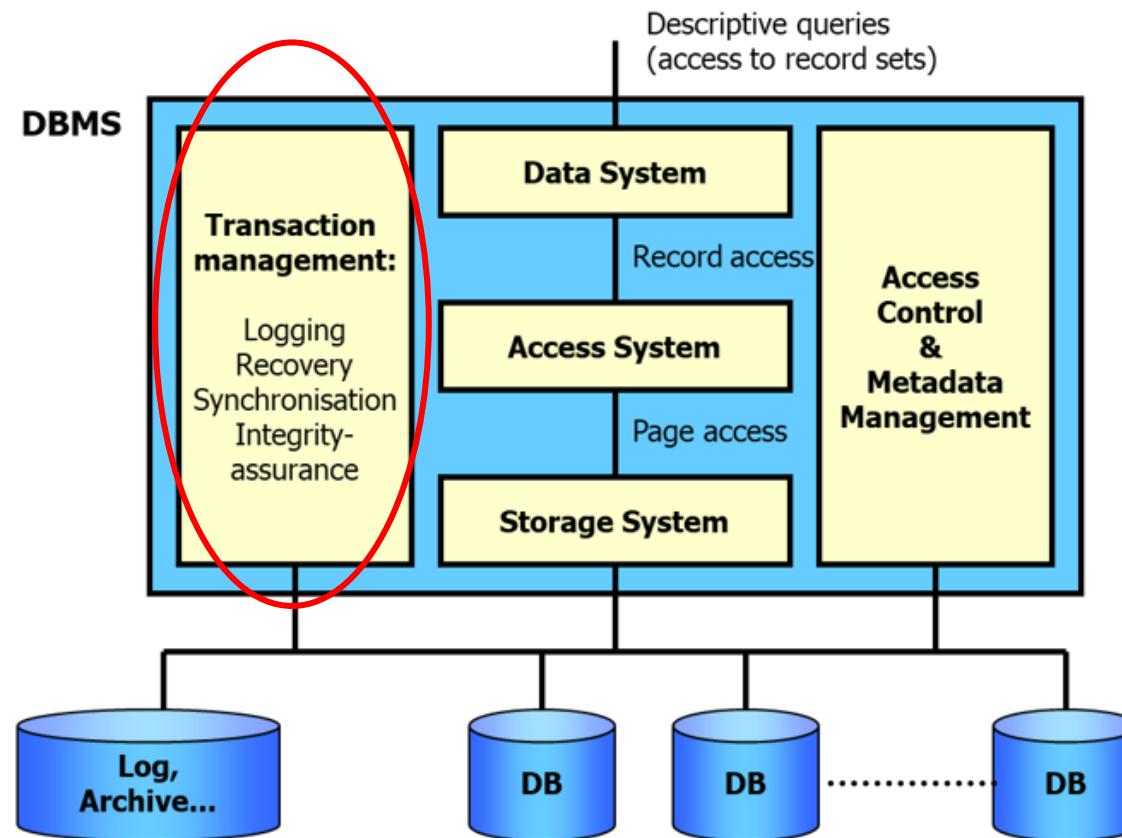
- **Relational layer:**
T1/T2 in conflict (*lost updates*), if tuples are modified simultaneously in the same block
⇒ *Correctness lost due to physical realization*
- **File layer:**
If T1 and T2 (logically independent) change tuples in same page, they must be serialized on file layer
⇒ *Inefficient due to unnecessary locks*

Transaction management can be considered as part of file layer (loss of efficiency) or as part of segment layer (a little bit better), but:

Correctness and full performance require comprehensive Transaction Management



Back to the Layered Architecture



[Härder & Rahm, 2001]

Transaction

Definition 6.1:

A ***transaction*** (TX) is a DB program, which only consists of *read and write operations* to a database. These operations are denoted as *read(x)* or *write(x)*, where x is a DB object.

- The restriction on read and write operations is an abstraction
- Thus, semantic information about the DB program is lost.
- This abstraction is sufficient for many practical applications.
- This abstraction is often too rough (multiple layer transactions)



6.1 Synchronization Problems

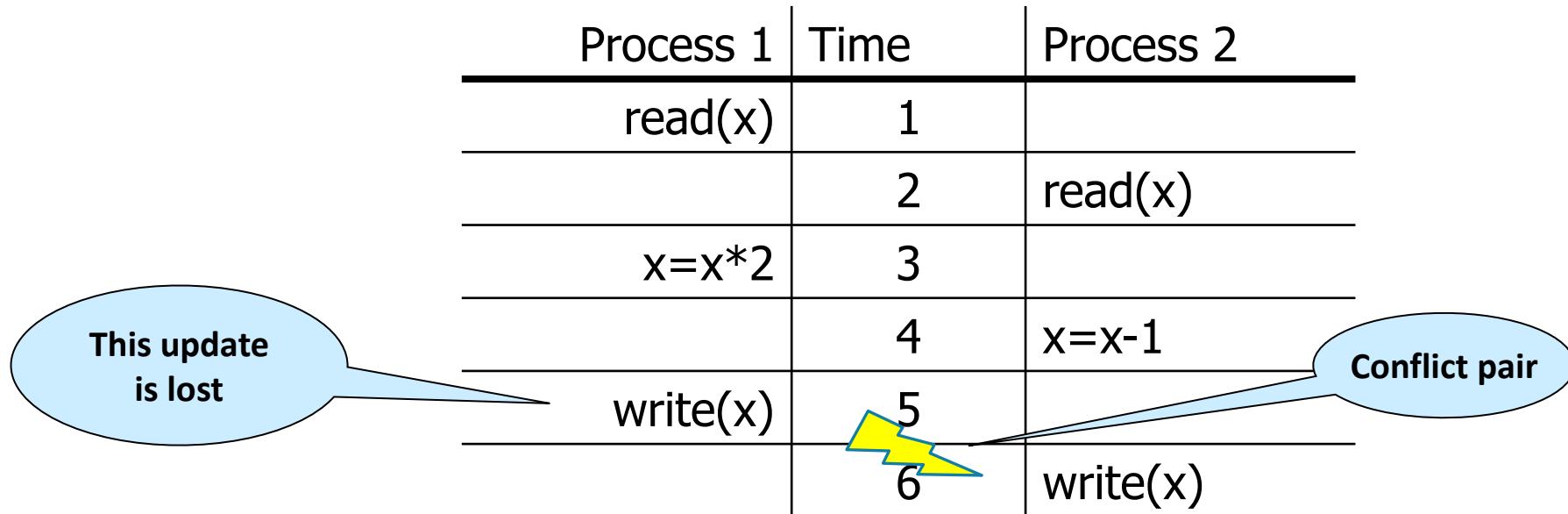
Learning Goals

At the end of this section you will be able to

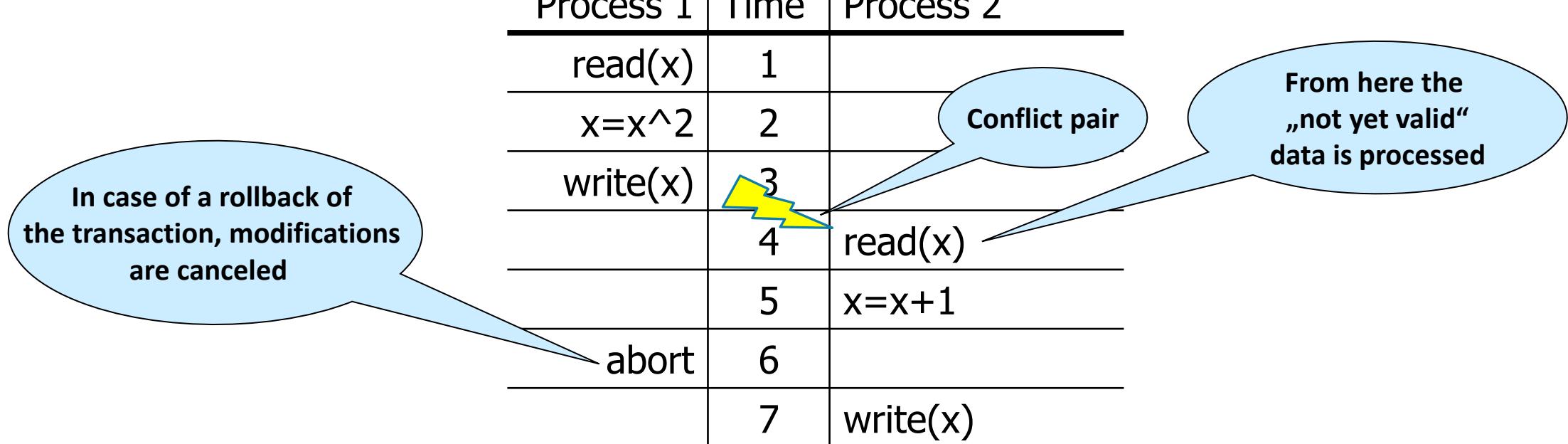
- ✓ name and explain synchronization problems
- ✓ name and explain the ACID properties
- ✓ explain what that means for the implementation in the TX management



Lost Update



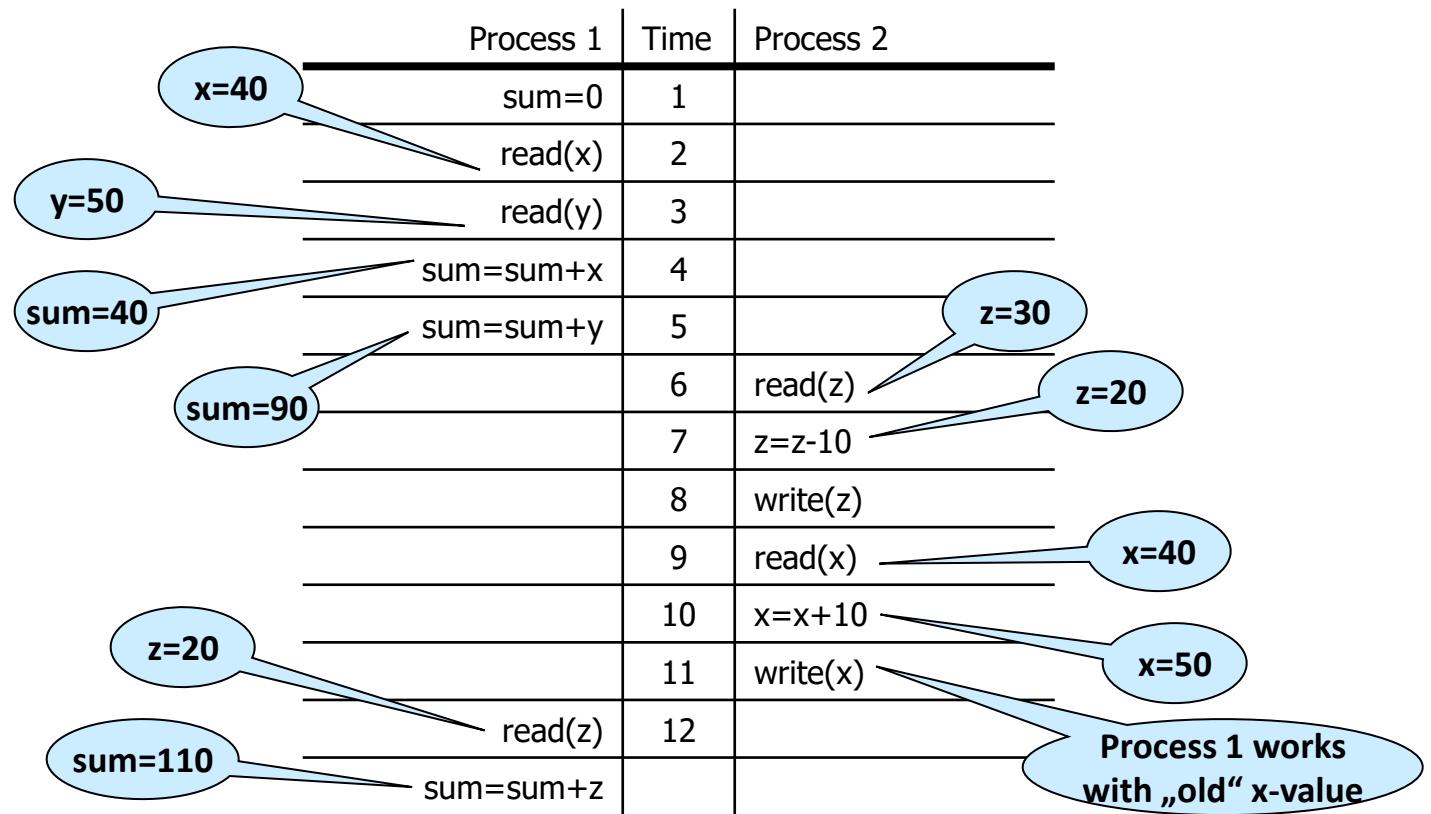
Dirty Read



⇒ For the synchronization of processes, conflict pairs must be identified and handled

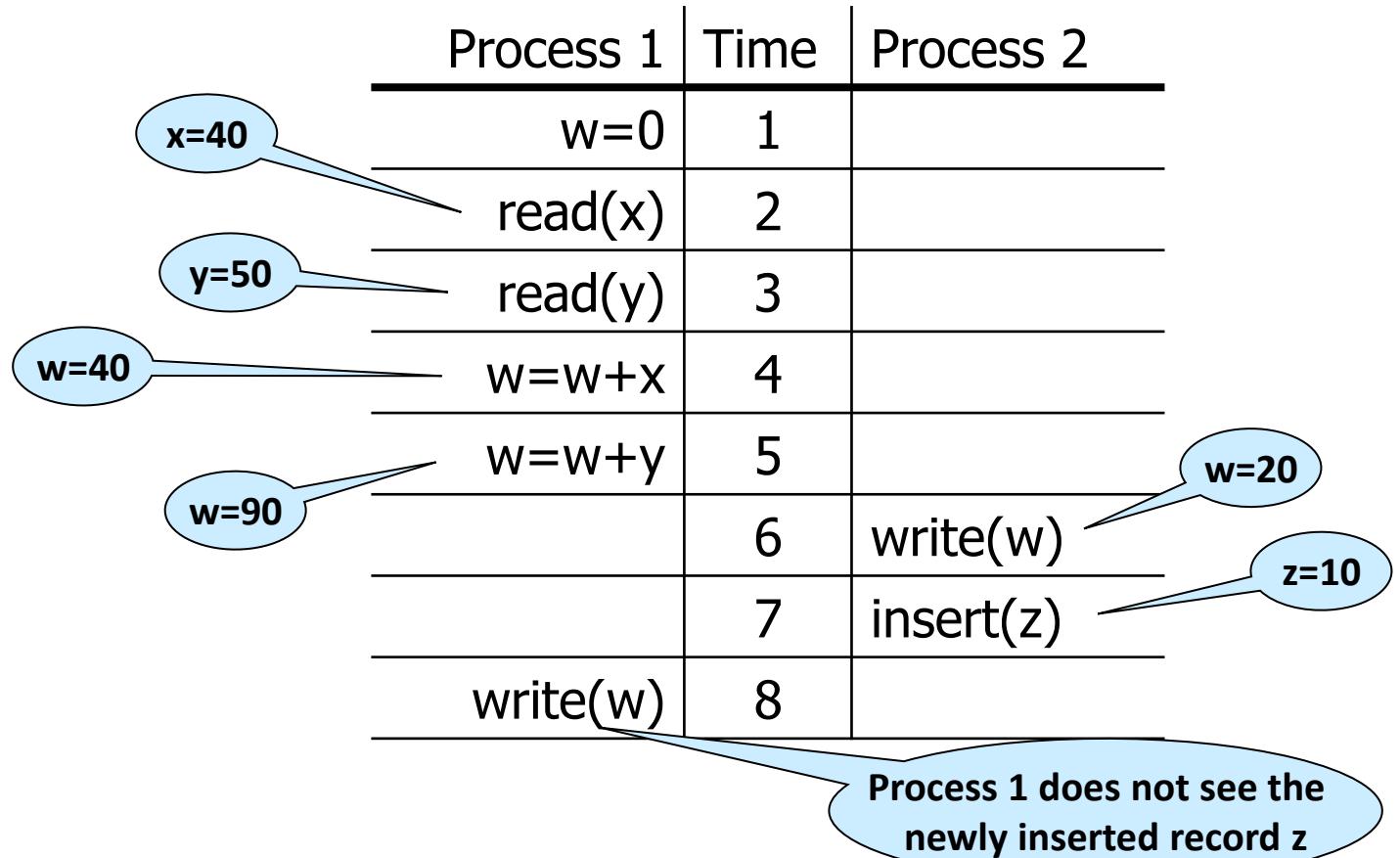
Non-Repeatable-Read / Inconsistent-Read

- Given: $x=40$, $y=50$, $z=30$
- Process 1: calculates the current sum
- Process 2: transfers 10 from z to x



Phantom Problem

- Table T containing records $x=40$, $y=50$
- Process 1:** calculates the sum of T and writes it to w (e.g., in another table)
- Process 2:** writes $w=20$ and inserts an element $z=10$ into T



ACID Properties

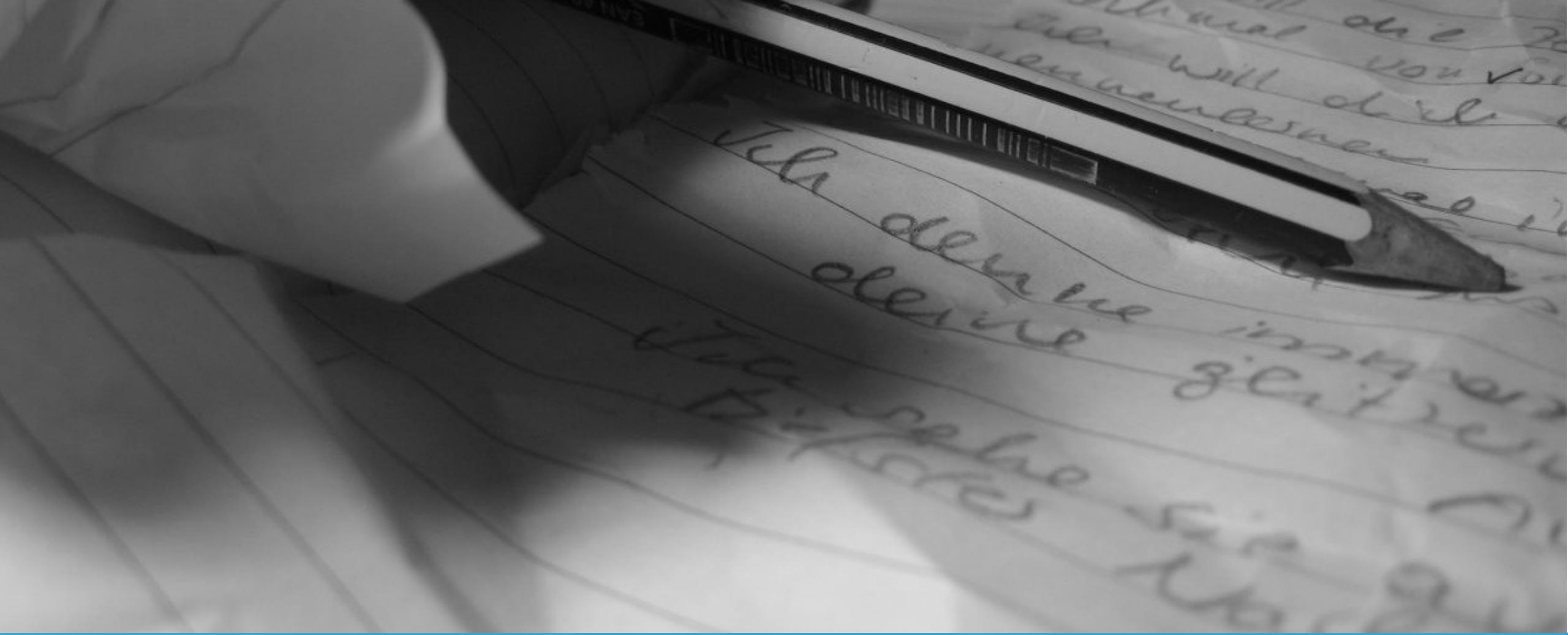
Atomicity	Consistency	Isolation	Durability
<ul style="list-style-type: none">In an execution of a transaction, either all operations are carried out, or none are.	<ul style="list-style-type: none">Preservation of all integrity constraints of the DB, i.e., a transaction starts with a consistent DB state, and after the execution of the transaction the DB state is consistent as well.	<ul style="list-style-type: none">Isolated execution of a transaction, i.e., „as if executed solely“	<ul style="list-style-type: none">Once a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

The ACID Principle

Every transaction must be processed in a way that the ACID properties are preserved.

Direct consequences from the ACID Principle:

- If a failure occurs during the execution of a transaction, i.e., the transaction cannot be completed successfully, it will leave ***no trace*** in the database.
- The database state is ***always well defined*** after error recovery.
- Isolation ***prevents anomalies*** in multi-user operation, if every single transaction completes without any error, and the atomicity and persistence of the system are guaranteed



6.2 The Read-Write Model

EvaSys Survey

10 min.



<https://evasys.rwth-aachen.de/evasys/online.php?pswd=XC2Z9>



Learning Goals

At the end of this section you will be able to

- ✓ formally define and write a transaction
- ✓ define and write a schedule



Read-Write Model

Abstracted basic assumption:

A database $D = \{x, y, z, \dots\}$ is a set of (database) objects, to which the read and/or write operations have access.

Definition 6.2:

Let $D = \{x, y, z, \dots\}$ be a database.

Then a ***transaction*** t (TX) is a finite series of operations in the form $r(x)$ („read x “) or $w(x)$ („write x “) denoted as

$$t = p_1, \dots, p_n$$

with $n < \infty$, $p_i \in \{r(x), w(x)\}$ for $1 \leq i \leq n$ and $x \in D$.

Indices are used to distinguish various (concurrent) transactions.

Example Transactions

$$D = \{x, y, z\}$$

$$t_1 = r_1(x) \ r_1(y) \ r_1(z) \ w_1(z) \ w_1(x)$$

$$t_2 = r_2(x) \ r_2(z) \ w_2(x)$$

Read-Write Model – Syntax and Semantics

- Formally, all the steps of two transactions are disjunctive ($t_i \cap t_j = \emptyset$)
- A DB object and an action are associated with each transaction step:
 - $p_i=r(x)$ Action p_i reads object x
 - $p_i=w(x)$ Action p_i writes object x
- Transactions are syntactical constructs (their semantics are unknown)

Read-Write Model – Abstract Semantics

Actions of a transaction t can be interpreted as follows

- The initial value of an object x is x_0
- If $p_i = r(x)$, a program variable v_i is assigned the current value of x .
- If $p_i = w(x)$, x is assigned a value v_i , which is calculated by the program.
- Each of these values written by a TX t depend on all values, which have been read by t so far:

$$x = f_{tx}(v_{j_1}, \dots, v_{j_k}) \text{ with } \{j_1, \dots, j_k\} = \{j_r \mid p_{j_r} = r(\cdot) \wedge j_r < i\}$$

Example

$D=\{x, y, z\}$

Given a transaction

$$t=r(x) \ r(y) \ w(z) \ r(z) \ w(x)$$

thus

$$p_1=r(x), p_2=r(y), p_3=w(z), p_4=r(z), p_5=w(x)$$

moreover

$$z = f_{tz}(v_1, v_2) = f_{tz}(x_0, y_0)$$

$$x = f_{tx}(v_1, v_2, v_4) = f_{tx}(x_0, y_0, f_{tz}(x_0, y_0))$$

Schedules

Definition 6.3:

Let $T = \{t_1, \dots, t_n\}$ be a (finite) set of transactions. Thus

- ***shuffle*(T)** is the *Shuffle Product* of T .
(The set of all permutations of all actions in t_1, \dots, t_n and no other actions except these.)
- A ***complete schedule*** s for T is a series $s' \in \text{shuffle}(T)$ with the additional pseudo actions c_i (*commit*) and a_i (*abort*) for each $t_i \in T$ according to the following rules:
 1. $(\forall i, 1 \leq i \leq n) c_i \in s \Leftrightarrow a_i \notin s$
 2. $(\forall i, 1 \leq i \leq n) c_i$ or a_i are in s , wherever, but after the last action of t_i
- $\text{shuffle}_{ac}(T)$ is the set of all complete schedules
- A ***schedule*** is a prefix of a complete schedule
- A complete schedule is ***serial***, if for a permutation ρ from $\{1, \dots, n\}$ it holds: $s = t_{\rho(1)} \dots t_{\rho(n)}$

Notations for Schedule s

$\text{trans}(s)$ = $\{t_i \mid s \text{ contains actions of } t_i\}$

$\text{commit}(s)$ = $\{t_i \in \text{trans}(s) \mid c_i \in s\}$

$\text{abort}(s)$ = $\{t_i \in \text{trans}(s) \mid a_i \in s\}$

$\text{active}(s)$ = $\text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s))$

$\text{op}(s)$ = set of all the actions occurring in s



6.3 Serializability

Learning Goals

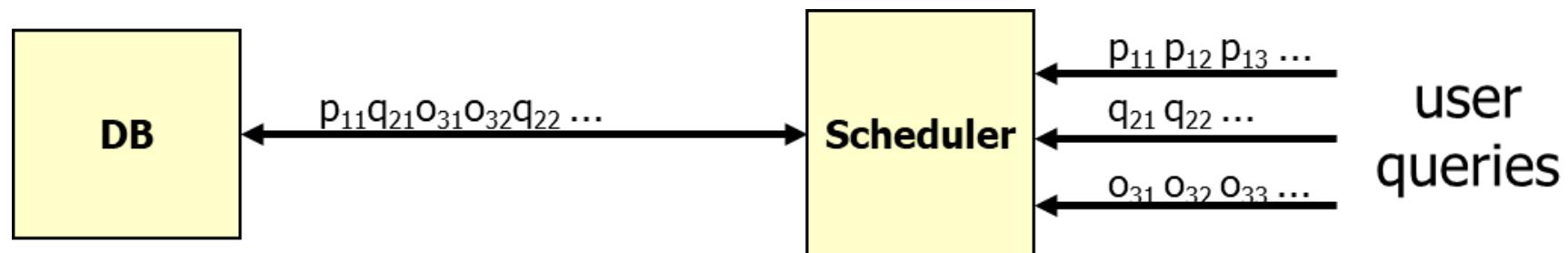
At the end of this section you will be able to

- ✓ explain what means serializability
- ✓ formally define and explain different kinds of serializability concepts
- ✓ determine when transactions are in conflict in a schedule
- ✓ determine if a schedule can be serialized and in which serializability class a schedule is



Idea

- If ACID transactions are performed sequentially, the consistency of the database is preserved.
- Hence, all serial schedules of ACID transactions are *correct*.
- If transactions are performed concurrently, as if they **would** be executed serially, we name them ***serializable*** schedules
- The „pseudo-serialization“ of concurrent queries is the task of ***schedulers***



Correctness of Schedules

- S is the set of all schedules
- **Challenge:** define subset of S which contains only *correct* schedules (delivers a correct end result), denoted by $\text{correct}(S)$
- **Requirements**
 - $\text{correct}(S) \neq \emptyset$
 - $s \in \text{correct}(S)$ is efficiently decidable
 - $\text{correct}(S)$ is sufficiently large
 - Many possibilities to generate a correct schedule
 - Higher degree of concurrency

Final-State Serializability

Definition 6.4

Let s and s' be schedules. s and s' are called ***final-state equivalent***, denoted as

$$s \approx_f s',$$

if $\text{op}(s) = \text{op}(s')$ and all DB objects have ***at the end*** identical values in s and s' , according to the abstract semantics (slide 22).

A schedule s is called ***final-state serializable*** if there exists a serial schedule s' which is final-state equivalent to s .

FSR is the class of all final-state serializable schedules.

Example

- Each of these values written by a TX t depend on all values, which have been read by t so far:

$$x = f_{tx}(v_{j_1}, \dots, v_{j_k}) \text{ with } \{j_1, \dots, j_k\} = \{j_r \mid p_{j_r} = r(\cdot) \wedge j_r < i\}$$

$s = r_1(x) r_2(y) w_1(y) r_3(z) w_3(z) r_2(x) w_2(z) w_1(x)$

$s' = r_3(z) w_3(z) r_2(y) r_2(x) w_2(z) r_1(x) w_1(y) w_1(x)$

Example

- Final-state equivalence can not be determined merely by the last („final“) write operation
- In addition, the previous write operations must also be taken into consideration

Let $s = r_1(x) r_2(y) w_1(y) w_2(y) c_1 c_2$ and

$s' = r_1(x) w_1(y) r_2(y) w_2(y) c_2 c_1$.

Thus:

In s : $y=f_{2y}(y_0)$

In s' : $y=f_{2y}(f_{1y}(x_0))$

→ $s \not\approx_f s'$.

$w_i(x)$ does not occur; therefore, $x=x_0$ in both schedules.

Reads-from Relation

Definition 6.5

- Given is a schedule s , extended with an initial and a final transaction, t_0 and t_∞ .
- $r_j(x)$ **reads** x in s **from** $w_i(x)$ if $w_i(x)$ is the last write on x , i.e., $w_i(x) <_s r_j(x)$
- **reads-from relation** of s is $RF(s) := \{(t_i, x, t_j) \mid \text{an } r_j(x) \text{ reads } x \text{ from a } w_i(x)\}$
- Operation p is **directly useful** for operation q , denoted as $p \rightarrow q$, if q reads from p , or p is a read operation and q is a subsequent write operation of the same transaction
- The **useful relation**, denoted as \rightarrow^* , is the reflexive and transitive closure of \rightarrow
- Operation p is called **alive** in s , if it is useful for some operation from t_∞ , and **dead** otherwise.
- The **live-reads-from relation** of s is denoted as
$$LRF(s) := \{(t_i, x, t_j) \mid \text{an alive } r_j(x) \text{ reads } x \text{ from } w_i(x)\}$$

Example

$s = r_1(x) \ r_2(y) \ w_1(y) \ w_2(y) \ c_1 \ c_2$

$s' = r_1(x) \ w_1(y) \ r_2(y) \ w_2(y) \ c_2 \ c_1$

$\text{RF}(s) = \{(t_0, x, t_1), (t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$

$\text{RF}(s') = \{(t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$

Consequently:

$\text{LRF}(s) = \{(t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$

$\text{LRF}(s') = \{(t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty)\}$

For s and s' is valid:

$w_0(x) \rightarrow r_1(x) \rightarrow w_1(y)$,

$w_0(y) \rightarrow r_2(y) \rightarrow w_2(y) \rightarrow r_\infty(y)$, thus also

$w_0(x) \rightarrow^* w_1(y)$,

$w_0(y) \rightarrow^* r_\infty(y)$

In s' additionally

$w_1(y) \rightarrow r_2(y)$, thus

$w_1(y) \rightarrow^* r_\infty(y)$ and even

$r_1(x) \rightarrow^* r_\infty(y)$

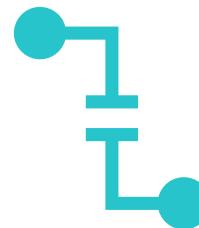
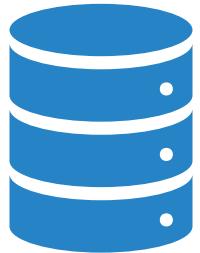
$r_1(x), w_1(y)$
are dead

Theorem 6.1:

Let s and s' be schedules.

Thus valid $s \approx_f s' \Leftrightarrow \text{LRF}(s) = \text{LRF}(s')$

Problems with Final-State Serializability



Testability

Final-State Serializability can only be checked at the “end” of the database

→ We need a serializability concept that can be tested at any time!

Complexity

There are $n!$ serial schedules.

Checking for Final-State Serializability has exponential complexity

How About the Anomalies? – Lost Update

$$L = r_1(x) \ r_2(x) \ w_1(x) \ w_2(x) \ c_1 \ c_2$$

$$LRF(L) = \{(t_0, x, t_2), (t_2, x, t_\infty)\}$$

$$LRF(t_1 t_2) = \{(t_0, x, t_1), (t_1, x, t_2), (t_2, x, t_\infty)\}$$

$$LRF(t_2 t_1) = \{(t_0, x, t_2), (t_2, x, t_1), (t_1, x, t_\infty)\}$$

Process 1	Time	Process 2
read(x)	1	
	2	read(x)
x=x*2	3	
	4	x=x-1
write(x)	5	
	6	write(x)



Not in FSR

Inconsistent Read Anomaly

$$I = r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2$$

$$LRF(I) = \{(t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$

$$LRF(t_1t_2) = \{(t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$

$$LRF(t_2t_1) = \{(t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$



Is in FSR!

View Serializability

View serializability defines a first adaption of FSR, which is motivated as follows:

- If all participating TX are of type „read-only“ → beginning state = end state; hence, the FSR criterion is trivial
- FSR emphasizes „living“ operations → but each TX should read the identical value in two equivalent schedules, i.e., have the same „view“ on the data, independent of living / dead actions

Definition 6.6:

Let s, s' be two schedules. s and s' are called **view equivalent**, denoted as $s \approx_v s'$, if $RF(s) = RF(s')$, i.e., $s \approx_v s' \Leftrightarrow RF(s) = RF(s')$

A complete schedule is called **view serializable**, if a serial schedule s' (with $op(s) = op(s')$) exists with $s \approx_v s'$.

VSR is the class of view serializable schedules. It holds $VSR \subseteq FSR$ (even $VSR \subset FSR$).

But: it is NP-complete to determine if s belongs to VSR

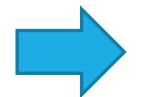
Inconsistent Read Anomaly

$$I = r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)c_1c_2$$

$$RF(I) = \{(t_0, x, t_2), (t_2, x, t_1), (t_0, y, t_1), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$

$$RF(t_1t_2) = \{(t_0, x, t_1), (t_0, y, t_1), (t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$

$$RF(t_2t_1) = \{(t_0, x, t_2), (t_0, y, t_2), (t_2, x, t_1), (t_2, y, t_1), (t_2, x, t_\infty), (t_2, y, t_\infty)\}$$



Is not in VSR!

Conflicts & Conflict Relations

Definition 6.7:

Let s be a schedule, $t, t' \in \text{trans}(s)$ and $t \neq t'$:

- Two data operations $p \in t$ and $q \in t'$ in s are in ***conflict***, if they operate on the same object **and** at least one of them is a write operation.
- $C(s)=\{(p,q) \mid p, q \text{ in } s \text{ are in conflict and } p \text{ is before } q \text{ in } s\}$ are the ***conflict relations*** of s .

Example

Let $s = w_1(x) r_2(x) w_2(y) r_1(y) w_1(y) w_3(x) w_3(y) c_1 a_2$

Then: $C(s) = \{ (w_1(x), r_2(x)), (w_1(x), w_3(x)), (r_2(x), w_3(x)),$
 $(w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_2(y), w_3(y)),$
 $(r_1(y), w_3(y)), (w_1(y), w_3(y)) \}$

Note:

$\text{conf}(s)$ denotes in the following the ***conflict relations*** of a schedule s , which are cleaned up by aborted transactions.

$\text{conf}(s) = \{ (w_1(x), w_3(x)), (r_1(y), w_3(y)), (w_1(y), w_3(y)) \}.$



Conflict Equivalence

Example:

Let $s = r_1(x) r_1(y) w_2(x) w_1(y) r_2(z) w_1(x) w_2(y)$,
 $s' = r_1(y) r_1(x) w_1(y) w_2(x) w_1(x) r_2(z) w_2(y)$.
Hence $s \approx_c s'$.

Definition 6.8:

Let s and s' be two schedules. s and s' are called ***conflict equivalent***, denoted as $s \approx_c s'$, if:

- $\text{op}(s)=\text{op}(s')$ and
- $\text{conf}(s)=\text{conf}(s')$.

Can be tested easily: For two given schedules (with the same sets of operations)
determine the conf relations and test their equality.

Conflict Serializability

Example:

$s = r_2(y) \text{ } w_1(x) \text{ } w_1(y) \text{ } c_1 \text{ } w_2(x) \text{ } c_2$ $\notin \text{CSR}$ $\text{conf}(s) = \{(r_2(y), w_1(y)), (w_1(x), w_2(x))\}$

$\text{conf}(t_1 t_2) = \{(w_1(x), w_2(x)), w_1(y), r_2(y)\}$

$\text{conf}(t_2 t_1) = \{(r_2(y), w_1(y)), (w_2(x), w_1(x))\}$

$s' = r_1(x) \text{ } r_2(x) \text{ } w_2(y) \text{ } c_2 \text{ } w_1(x) \text{ } c_1$ $\in \text{CSR}$ $\text{conf}(s') = \{ (r_2(x), w_1(x)) \}$

$\text{conf}(t_2 t_1) = \{ (r_2(x), w_1(x)) \}$

Conflict Serializability

Definition 6.9:

A complete schedule s is called ***conflict serializable***, if a serial schedule s' exists with $s \approx_c s'$.

CSR is the class of all conflict serializable schedules.

Theorem 6.2:

It holds: $\text{CSR} \subset \text{VSR} \subset \text{FSR}$

Important: Membership in CSR can be tested easily with a conflict graph.

Conflict Graph

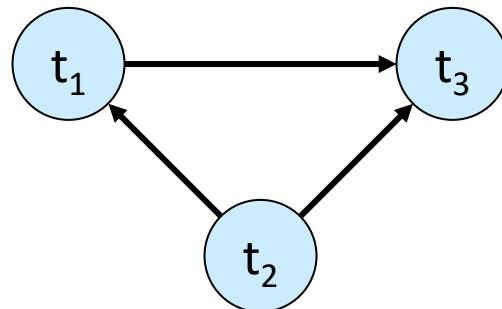
Definition 6.10:

Let s be a schedule s . The **conflict graph** $G(s)=(V,E)$ of s is defined by $V=\text{commit}(s)$ and

$$(t,t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t) (\exists q \in t') (p,q) \in \text{conf}(s)$$

Example:

$s = r_1(x) r_2(x) w_1(x) r_3(x) w_3(x) w_2(y) c_3 c_2 w_1(y) c_1$



Conflict Serializability Theorem

Theorem 6.3 (Serializability):

$$s \in \text{CSR} \Leftrightarrow G(s) \text{ is acyclic}$$

Proof: $s \in \text{CSR} \Rightarrow G(s)$ is acyclic

- Let $s \in \text{CSR}$, then there exists a serial schedule s' with $\text{op}(s)=\text{op}(s')$ and $\text{conf}(s)=\text{conf}(s')$, i.e., $s \approx_c s'$.
- If consequently $t, t' \in V$ with $t \neq t'$ and $(t,t') \in E$ for $G(s)=(V,E)$, thus:
 $(\exists p \in t) (\exists q \in t') p$ is before q in s and $(p,q) \in \text{conf}(s)$.
- Because the conflict relations of s and s' are identical, p is also before q in s' .
- Because s' is serial, then the whole t is before the whole t' in s' .
- If $G(s)$ is cyclic, there exists a cycle of the form $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k \rightarrow t_1$.
- The same cycle exists then in s' , a discrepancy to the serializability of s' .

Proof: $s \in \text{CSR} \Leftarrow G(s)$ is acyclic

- If $G(s) = (V, E)$ with $V = \{t_1, \dots, t_n\}$ is acyclic, thus $G(s)$ is topologically sorted.
- The result is then $t_{\rho(1)} \dots t_{\rho(n)} = s'$ for a permutation ρ . s' is serial
- We prove $s \approx_c s'$:
 - If $p \in t, q \in t'$ for $t, t' \in V$ with p and q in s and $(p, q) \in \text{conf}(s)$, consequently $(t, t') \in E$.
 - Then in the topological sort t is before t' , i.e., in s' p is before q too.
 - Because both operations are in conflict, consequently $\text{conf}(s) = \text{conf}(s')$.

□

Conflict Serializability – Membership Test

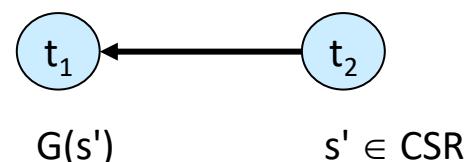
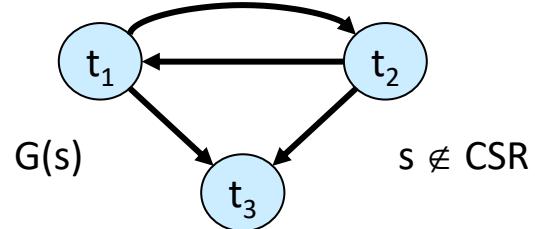
Corollary 6.1:

Membership in CSR can be tested in polynomial time.

Example Membership Test

Let $s = r_1(y) r_3(w) r_2(y) w_1(y) w_1(x) w_2(x) w_2(z) w_3(x)$
 $s' = r_1(x) r_2(x) w_2(y) w_1(x)$

Corresponding conflict graphs:





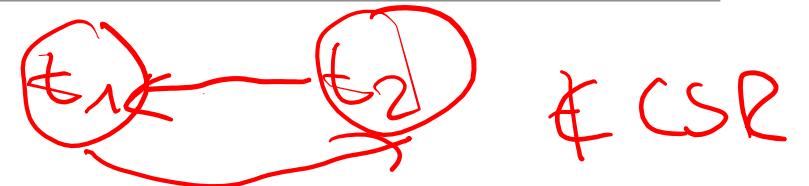
Quiz

Is this schedule in CSR?

$$s = r_1(y) w_1(y) w_3(x) w_1(x) \\ r_2(y) r_3(z) w_2(x) w_2(z)$$

Membership Test - Anomalies

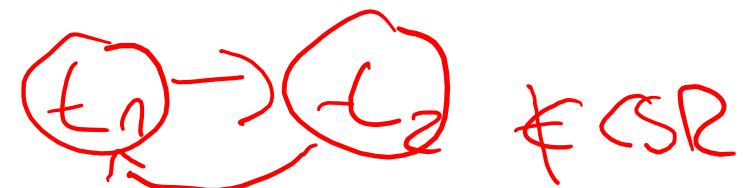
Lost Update: $s = r_1(x) \underline{r_2(x)} w_1(x) c_1 \underline{w_2(x)} c_2$



Dirty Read: $s = r_1(x) \underline{w_1(x)} r_2(x) a_1 \underline{w_2(x)} c_2$



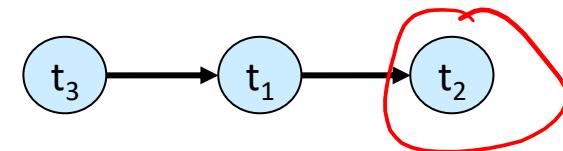
Inconsistent Read: $s = r_1(x) \underline{r_1(y)} r_2(z) \underline{w_2(z)} r_2(x) \underline{w_2(x)} c_2 r_1(z) c_1$



Order-Preserving Conflict Serializability

Consider

$$s = w_1(x) \ r_2(x) \ c_2 \ w_3(y) \ c_3 \ w_1(y) \ c_1$$



Thus: $s \in \text{CSR}$ since $s \approx_C t_3 t_1 t_2$.

But: t_2 is already terminated, before t_3 starts. This is not expected.

Solution: Define constraint on CSR, which requires that transactions that do not overlap in time, appear in the same order in a conflict equivalent schedule

Order-Preserving Conflict Serializability

Definition 6.11:

A complete schedule s is called ***order-preserving conflict serializable***, there exists a serial schedule s' with $s \approx_c s'$ and the following holds for all $t, t' \in \text{trans}(s)$:

If t occurs completely before t' in s , then the same holds in s' .

OCSR is the class of all order-preserving conflict serializable schedules

$\text{OCSR} \subset \text{CSR}$, e.g., $s = w_1(x) \ r_2(x) \ c_2 \ w_3(y) \ c_3 \ w_1(y) \ c_1 \in \text{CSR-OCSR}$

Commitment Ordering

Another constraint on CSR is based on the observation, that for transactions in conflict, an ordering of their commits in „conflict order“ is sufficient for conflict serializability.

Definition 6.12:

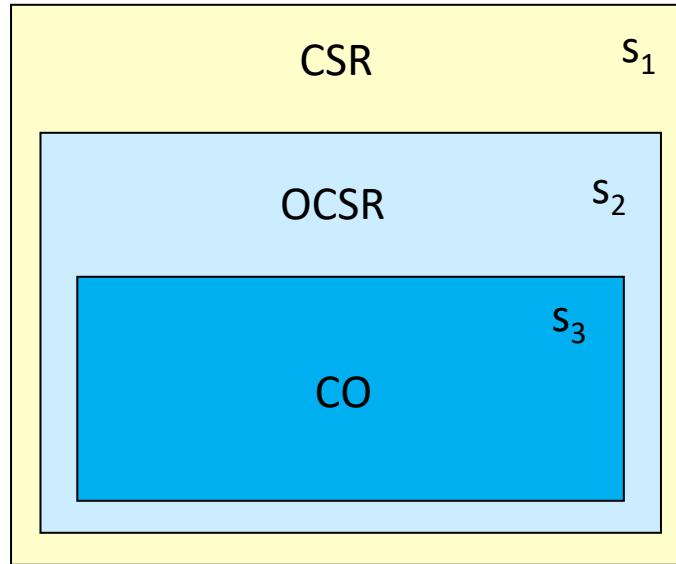
A schedule s is called ***commit order-preserving conflict serializable*** (or owns the property of ***commit order preservation***), if the following holds:

For all $t_i, t_j \in \text{commit}(s)$, $i \neq j$, with $(p, q) \in \text{conf}(s)$ for $p \in t_i, q \in t_j$, then: c_i is before c_j in s .

CO is the class of all schedules, which own the property.

It holds $\text{CO} \subset \text{CSR}$, e.g. $s = r_1(x) w_2(x) c_2 c_1 \in \text{CSR} - \text{CO}$

Summary CSR, OCSR, CO



with

- $s_1 = w_1(x) r_2(x) c_2 w_3(y) c_3 w_1(y) c_1 \in \text{CSR} - \text{OCSR}$
- $s_2 = w_3(y) c_3 w_1(x) r_2(x) c_2 w_1(y) c_1 \in \text{OCSR} - \text{CO}$
- $s_3 = w_3(y) c_3 w_1(x) r_2(x) w_1(y) c_1 c_2 \in \text{CO}$



6.4 Transaction Recovery

Learning Goals

At the end of this section you will be able to

- ✓ know and explain the different types of recovery classes for a schedule
- ✓ determine for a schedule in which recovery class it is in
- ✓ explain which problems are avoided by schedules from which class



Problem of Robustness

Central question: Under which conditions does a schedule allow a correct recovery of transactions?

- Serializability alone does not avoid synchronization problems
- Recovery properties of schedules are orthogonal to serializability
- „Proper“ positioning of termination operations is important
- Failure safety can be formally specified, and can be easily realized in practice

Example - Transaction Recovery

$$s = r_1(x) \ w_1(x) \ \underline{r_2(x)} \ \underline{a_1} \ w_2(x) \ c_2$$


It holds:

- $\text{conf}(s) = \text{conf}(r_2(x) \ w_2(x) \ c_2) = \text{conf}(t_2)$, i.e., $s \in \text{CSR}$
- s contains a Dirty Read situation. Therefore, s is not correct!

In the following „ $p <_s q$ “ means that action p in schedule s occurs before action q .

Recoverability

„Every transaction will not be released, until all other transactions from which it has read, are released.“

Definition 6.13:

A schedule s is called **recoverable**, if the following holds:

$$(\forall t_i, t_j \in \text{trans}(s), i \neq j) t_i \text{ reads from } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

RC is the class of all recoverable schedules.

Example:

Let $s_1 = w_1(x) \underline{w_1(y)} r_2(u) \underline{w_2(x)} \underline{r_2(y)} w_2(y) c_2 w_1(z) c_1$ notin RC

It holds: t_2 reads y from t_1 and $c_2 \in s$, but $c_1 <_{s_1} c_2$. Consequently $s_1 \notin \text{RC}$.

Recoverability – Further Example

Let $s_2 = w_1(x) \underline{w_1(y)} r_2(u) w_2(x) \underline{r_2(y)} w_2(y) w_1(z) c_1 c_2$

It holds: $s_2 \in RC$, because $c_1 <_{s_2} c_2$.

But: an abort of t_1 leads to the abort of t_2 .

This may give rise to ***cascading aborts*** (which is unwanted).

Avoidance of Cascading Aborts

„A transaction is only allowed to read values from already successfully completed transactions.“

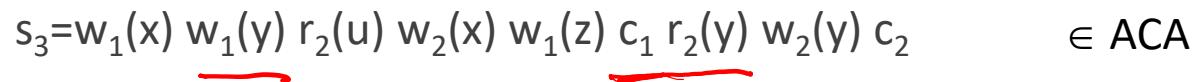
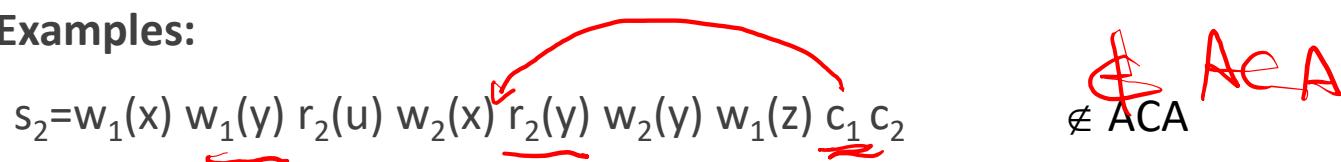
Definition 6.14:

A schedule s **avoids cascading aborts**, if it holds:

$$(\forall t_i, t_j \in \text{trans}(s), i \neq j) t_i \text{ reads } x \text{ from } t_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

ACA is the class of all schedules, which avoid cascading aborts.

Examples:



Further Problem

- If we abort a transaction, we roll all operations of the transaction back
- Values restored after abort, may be different from the Before Images of the write operations of the aborting transactions.

state of data objects
before changes are
made by the TX

⇒ Rollback requires undo of aborted transactions **AND** redo of committed transactions

Strictness

„A schedule is strict, if an object is not read or overwritten, until the transaction, which has written it at last, is terminated.“

Definition 6.15:

A schedule s is called *strict*, if the following holds:

$$(\forall t_i \in \text{trans}(s))(\forall p_i(x) \in \text{op}(t_i), p \in \{r,w\}) \quad w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$$

ST is the class of all strict schedules

Examples:

$$s_3 = w_1(x) w_1(y) r_2(u) \underbrace{w_2(x)}_{\text{overwritten}} \underbrace{w_1(z)}_{\text{overwritten}} c_1 r_2(y) w_2(y) c_2 \notin ST$$

$$s_4 = \underbrace{w_1(x)}_{\text{written}} \underbrace{w_1(y)}_{\text{written}} r_2(u) \underbrace{w_1(z)}_{\text{written}} c_1 \underbrace{w_2(x)}_{\text{not yet read}} \underbrace{r_2(y)}_{\text{not yet read}} w_2(y) c_2 \in ST$$



Quiz

- Is this schedule strict (in ST) ?

$s = \underline{w_1(y)} \ w_2(x) \ w_1(z) \ \underline{r_2(y)} \ c_1 \ w_2(x) \ r_2(y) \ w_2(y) \ c_2$

$w_1(y) \leq_s r_2(y)$, but

$r_2(y) \leq_s c_1 \Rightarrow \text{not ST}$

not ACA

Rigorous Schedules

„A schedule is rigorous, if it is strict and no object x is overwritten, until all transactions, which have read x at last, are terminated.“

Definition 6.16:

A schedule s is called ***rigorous***, if it is strict and satisfies the following condition:

$$(\forall t_i, t_j \in \text{trans}(s)) r_j(x) <_s \underbrace{w_i(x)}_{\text{written}} \quad i \neq j \Rightarrow \underbrace{a_j <_s w_i(x)}_{\text{written}} \vee \underbrace{c_j <_s w_i(x)}_{\text{written}}$$

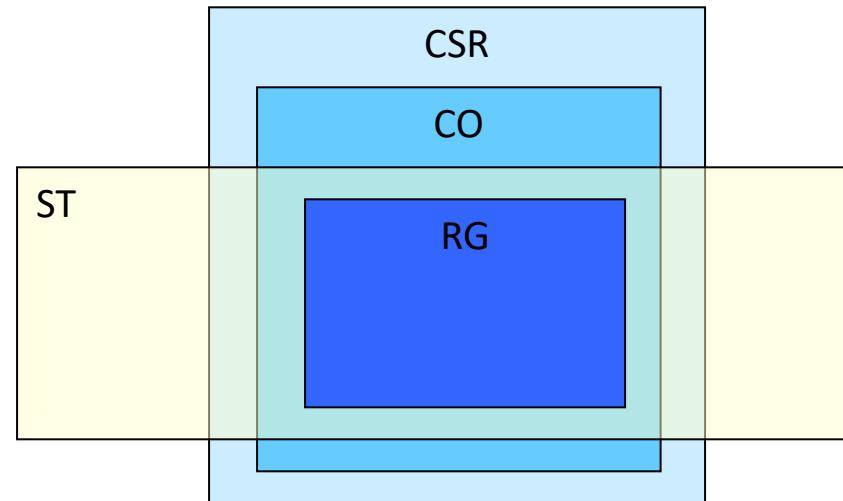
RG is the class of all rigorous schedules

Note

- A schedule from ST avoids Write-Read- as well as Write-Write conflicts between non-released transactions
- A schedule from RG avoids additionally Read-Write conflicts between those kinds of transactions.

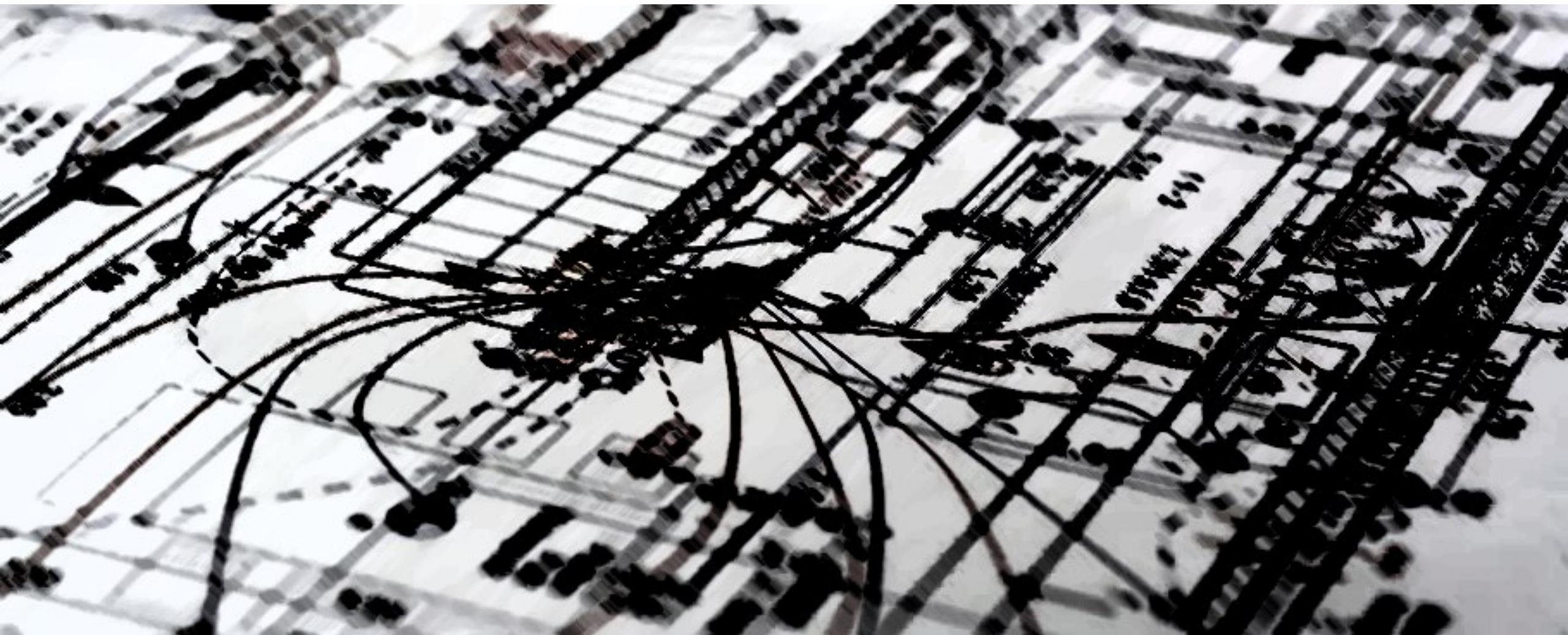
Therefore, $RG \subseteq ST$, but it holds also $RG \subset CSR$ and $RG \subset CO$

Relationships Serializability and Recovery Classes



Features of the class RG

- Their elements are not only in CSR but in ST as well, i.e., serializable and fault-safe and consequently correct
- We can check it without extra requirements (other than by CSR or by ST solely)



6.5 Concurrency Protocols

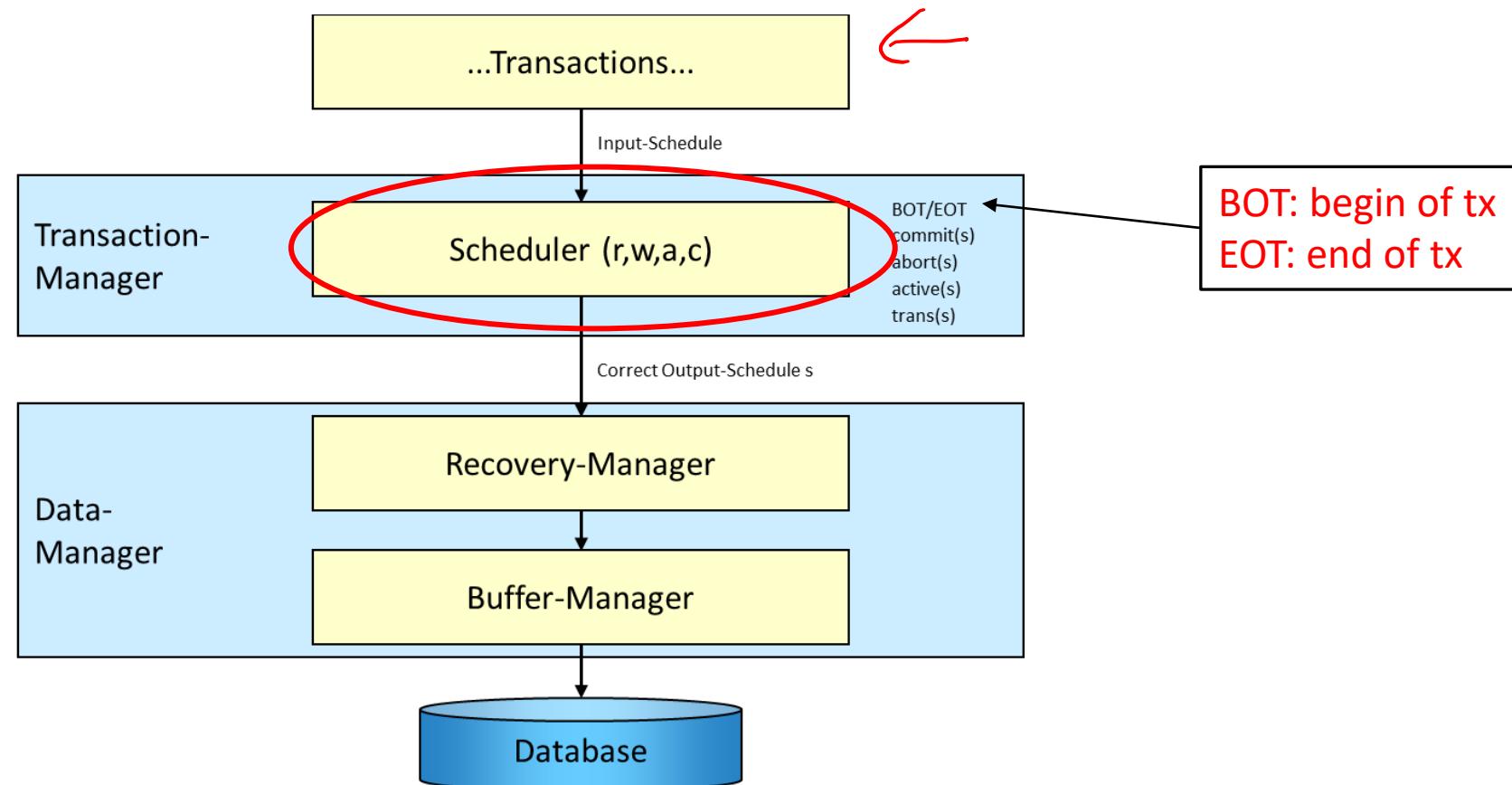
Learning Goals

At the end of this section you will be able to

- ✓ explain the task of a scheduler
- ✓ know different kinds of locks
- ✓ know and explain multiple classes of locking schedulers
- ✓ know and explain, which scheduler avoids which problems



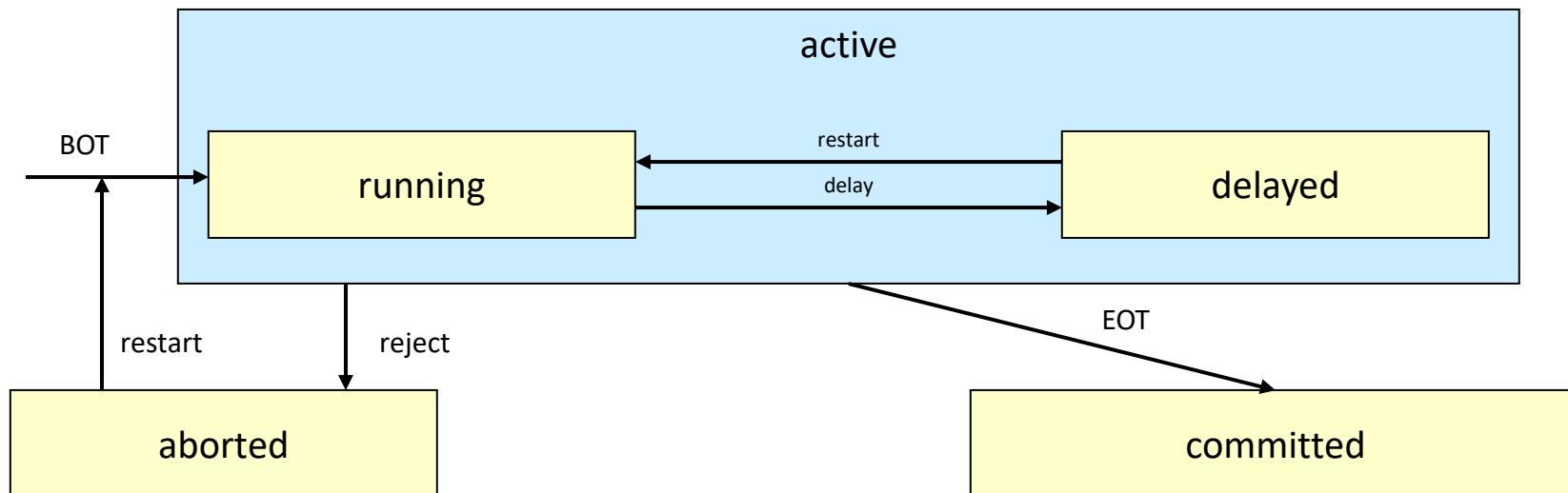
Transaction Processing Layers of a DBMS



Scheduling

- Techniques generating correct schedules for txs to be processed in a DBMS
- These are called ***scheduling protocols***, or in short ***scheduler***
- Two evaluation criteria of protocols
 - **Safety:** Is the output without exception in CSR or even in $\text{CSR} \cap \text{RC}$?
 - **Expressiveness:** Is a class of correct schedules completely or only partly generated?
- First classification: *locking* and *non-locking* schedulers

Scheduler Design - Possible States of a Transaction



Scheduler Design - Process

- Gets as input a sequence of operations r,w,a,c
- Must produce a correct output schedule from them

Three actions are possible per operation

1. Execute:

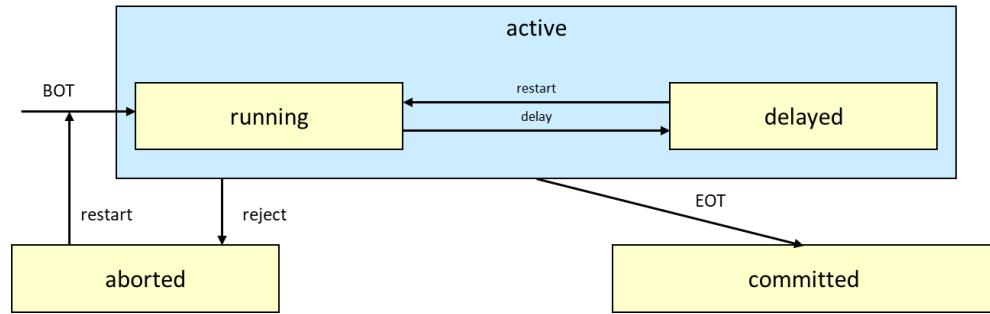
The operation is at once output, i.e., appended to the output schedule (r, w, a, or, c)

2. Reject:

The operation is not executed (r or w), abort of the concerning transaction

3. Delay:

The operation is neither executed nor rejected, but rather returned to the transaction manager (r or w)



Notations for Safety of a Scheduler S

- $\varepsilon(S)$: The result of S
- S is s-safe, if $\varepsilon(S) \subseteq \text{CSR}$
 - $\varepsilon(S)$ is serializable-safe
- S is f-safe, if $\varepsilon(S) \subseteq \text{RC}$ (or ACA or ST)
 - $\varepsilon(S)$ is failure-safe
- S is safe, if $\varepsilon(S) \subseteq \text{CSR} \cap \text{RC}$ (or ACA or ST)

Locking Scheduler - Concept

- Locks are applied for the synchronization of accesses on data objects that are used together
- These locks are set or removed by the scheduler for transactions
- Two types of locks for an object x:
 - Read lock (a.k.a. **shared lock** (sl)):
 $rl(x)$ (*read lock*)
 $ru(x)$ (*read unlock*)
 - Write lock (a.k.a. **exclusive lock** (xl)):
 $wl(x)$ (*write lock*)
 $wu(x)$ (*write unlock*)
- Locks in conflict \Leftrightarrow operations in conflict

Lock Compatibility		
	rl	wl
rl	+	-
wl	-	-

Rules for the Application of Locks

For each t_i , which is contained completely in a schedule s , the following should be valid:

1. If t_i contains a $r_i(x)$ [$w_i(x)$], $rl_i(x)$ [$wl_i(x)$] stands anywhere before it in s and $ru_i(x)$ [$wu_i(x)$] stands anywhere after it.
2. For each x processed by t_i there are exactly one $rl_i(x)$ resp. $wl_i(x)$ in s
3. No ru_i/wu_i is redundant

Examples:

$s_1 = rl_1(x) r_1(x) ru_1(x) wl_2(x) w_2(x) wl_2(y) w_2(y) wu_2(x) wu_2(y) c_2 wl_1(y) w_1(y) wu_1(y) c_1$

$s_2 = rl_1(x) r_1(x) wl_1(y) w_1(y) ru_1(x) wu_1(y) c_1 wl_2(x) w_2(x) wl_2(y) w_2(y) wu_2(x) wu_2(y) c_2$

Locking Protocols

For a schedule s , $DT(s)$ is the projection of s on the actions of the types r,w,a, and c
(**D**ata and **T**ermination operations), e.g.,

- $DT(s_1) = r_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1$ (\notin CSR)
- $DT(s_2) = r_1(x) w_1(y) c_1 w_2(x) w_2(y) c_2$ ($= t_1 t_2$)

Definition 6.17:

A scheduler *works according to a locking protocol*, if for every output s and every $t_i \in \text{trans}(s)$ it holds:

- t_i satisfies the rules from 1. to 3.
- If x is locked by t_i and t_j , where $t_i, t_j \in \text{trans}(s)$, $i \neq j$, then these locks are compatible

Two Phase Locking (2PL) - A First Concrete Scheduler

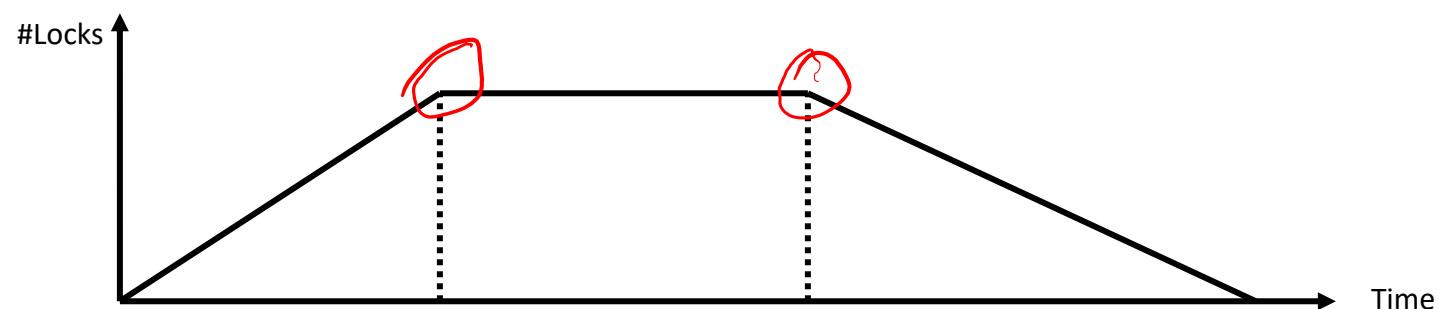
Definition 6.18:

A locking protocol is ***two phase***, if for every generated schedule s and every transaction $t_i \in \text{trans}(s)$ it holds:

After the first o_{t_i} action there is no further q_{t_i} action ($o, q \in \{r, w\}$).

Such a scheduler is called a ***2PL scheduler***.

In the first phase of a transaction, locks will only be set, in the second phase locks will only be removed.



Examples

$s_1 = r\l_1(x) \underline{r_1(x)} \underline{ru_1(x)} \underline{wl_2(x)} \underline{w_2(x)} \underline{wl_2(y)} \underline{w_2(y)} wu_2(x) wu_2(y) c_2 \underline{wl_1(y)} \underline{w_1(y)} wu_1(y) c_1$



s_1 violates the 2-Phase rules

$s_2 = r\l_1(x) \underline{r_1(x)} wl_1(y) \underline{w_1(y)} ru_1(x) \underline{wu_1(y)} c_1 \underline{wl_2(x)} \underline{w_2(x)} wl_2(y) \underline{w_2(y)} wu_2(x) wu_2(y) c_2$



s_2 is 2PL.

Safety of 2PL

committed
projection

1. s is 2PL generated; then for each $t_i \in CP(DT(s))$ it holds:

- If $o_i(x)$ ($\in \{r, w\}$) occurs in $CP(DT(s))$, then $ol_i(x)$ and $ou_i(x)$ occur also in it, and in the sequence

$$ol_i(x) < o_i(x) < ou_i(x)$$

- If $t_j \in CP(DT(s))$, $i \neq j$ and $p_i(x)$ and $q_j(x)$ are actions in conflict from $CP(DT(s))$, then it holds either

$$pu_i(x) < ql_j(x) \text{ or } qu_j(x) < pl_i(x).$$

- If $p_i(x)$ and $q_i(y)$ are in $CP(DT(s))$, then it holds: $pl_i(x) < qu_i(y)$.

Safety of 2PL

2. Let $G=G(\text{CP}(\text{DT}(s)))$ be the conflict graph of $\text{CP}(\text{DT}(s))$. Then it holds:

- If (t_i, t_j) is an edge in G , then it is valid $p_{i,j}(x) < q_{j,i}(x)$ for an object x and two operations $p_i(x), q_j(x)$ are thus in conflict.
- If (t_1, t_2, \dots, t_n) is a path in G , $1 \leq n$. Then it is valid that $p_{1,n}(x) < q_{n,1}(y)$ for two objects x and y and operations $p_1(x)$ and $q_n(y)$.
- G is acyclic.

Consequently

Theorem 6.2:

$$\varepsilon(2\text{PL}) \subseteq \text{CSR}$$

The true inclusion is even true!

Features of 2PL

Easy to implement

Better performance than other protocols

Easy to distribute

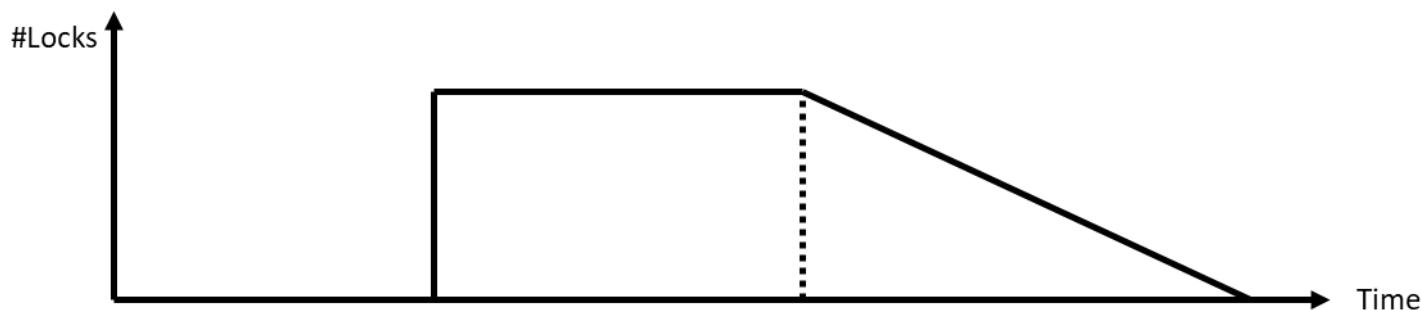
Not deadlock free, e.g., $r_1(x) w_2(y) w_2(x) c_2 w_1(y) c_1$

- Methods for deadlock detection
 - Transaction timer (Timestamps)
 - Waiting graphs
 - Deadlock prevention

Transactions may „starve“

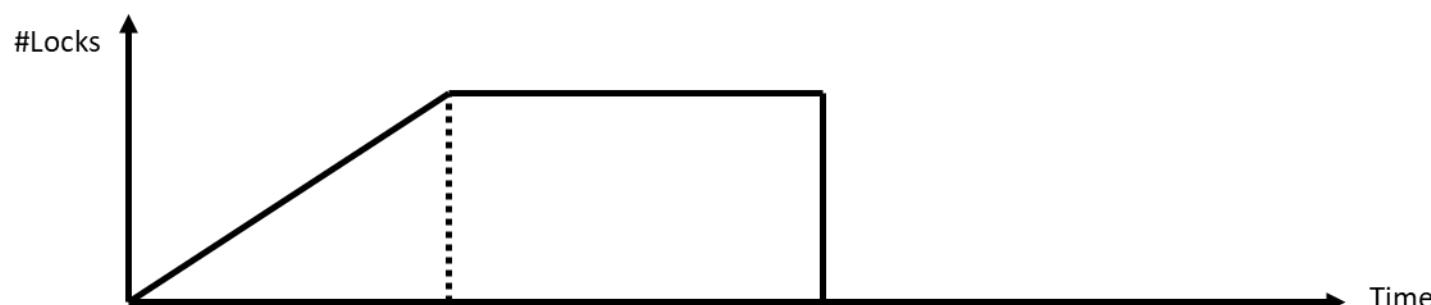
Variants of 2PL

- **Conservative 2PL:** All locks are available since BOT



- **Strict 2PL (S2PL):** All write locks are hold until EOT

- **Strong 2PL (SS2PL):** All locks are hold until EOT



Motivation for 2PL Variants

- When is it safe for the scheduler to remove a lock $ol_i(x)$ ($o \in \{r, w\}$)?
 1. When it knows, that t_i needs no lock any more
 2. When t_i does not access x later any more.
- Both preconditions are (at least) satisfied at the end of t_i .

Safety of S2PL Schedules

Proof:

Let s be a S2PL generated output; for $w_i(x) < o_j(x)$, $i \neq j$, $o \in \{r, w\}$ it holds then at first (because of 2PL):

i) $wl_i(x) < w_i(x) < wu_i(x)$ and

ii) $ol_j(x) < o_j(x) < ou_j(x)$



Because $wl_i(x)$ and $ol_j(x)$ are in conflict, follows either

(a) $wu_i(x) < ol_j(x)$ or

(b) $ou_j(x) < wl_i(x)$.



(b), (i), and (ii) contradicts the initial assumption $w_i(x) < o_j(x)$. Thus (a) is valid.

If the scheduler locks till EOT, it follows further

$$a_i < wu_i(x) \vee c_i < wu_i(x), \text{ i.e., } a_i < o_j(x) \vee c_i < o_j(x) \text{ and therefore } DT(s) \in ST.$$



Safety of S2PL Schedules

Theorem 6.4:

$\varepsilon(\text{S2PL}) \subseteq \text{CSR} \cap \text{ST}$, i.e., an S2PL scheduler is safe.

Disadvantages of 2PL Protocols

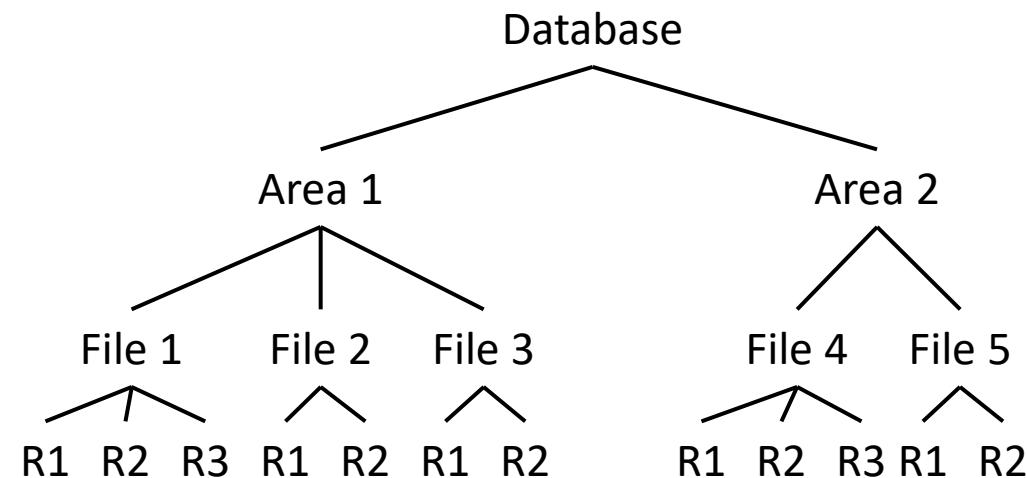
- If 2PL locking objects are „big“, then there are only a few locks to manage, but conflicts between locks occur more often
- If 2PL locking objects are „small“, then more concurrency is possible, but the costs for lock management are higher

Alternative locking protocols:

- 2PL generalization to generate exactly the class OCSR (O2PL)
 - Multiple granularity locking (MGL)
 - Tree locking (TL)

Multi-granularity Locking (MGL)

- Every transaction can choose the suitable granularity by itself.
(record, file, table space area, database).
- The scheduler must then prevent, that transactions set conflicting locks in overlapping granularities.



Multi-granularity Locking – Lock Types

If the database is tree-like structured, the following two provisions are helpful :

- Distinction between explicit and implicit locks
- Propagation of locks in tree upwards as *intentional locks* (irl, iwl, riwl)
- Compatibility of the 5 lock types:

	rl	wl	irl	iwl	riwl
rl	+	-	+	-	-
wl	-	-	-	-	-
irl	+	-	+	+	+
iwl	-	-	+	+	-
riwl	-	-	+	-	-

To Get	Need on all ancestors
irl or rl	irl or wrl
iwl, irwl, or wl	iwl or irwl

Multi-granularity Locking – Protocol

Each transaction t_i is locked/unlocked as follows:

1. If x is not the root of the database, then t_i must own a ir- or iw-lock on the parent node of x , in order to be able to set $rl_i(x)$ or $irl_i(x)$.
2. If x is not the root of the database, then t_i must own a iw-lock on the parent node of x , in order to be able to set $wl_i(x)$ or $iwl_i(x)$.
3. To read (write) x , t_i must own a r- or w-lock (w-lock) on x .
4. t_i cannot remove an intentional lock on x , as long as t_i has still a lock on a child of x .

That is, the locks are set top-down and are removed bottom-up.

We can prove, that for every transaction, which keeps the 2-Phase rules, $\varepsilon(MGL) \subseteq CSR$ is valid.



6.6 Index Locking

Learning Goals

At the end of this section you will be able to

- ✓ explain the advantages of index locking
- ✓ Explain and apply tree index locking algorithm



Assumption so far

- DB is a *fixed* collection of *independent* objects
- Even Strict 2PL might not guarantee serializability if objects are added

Example: (Phantom Problem, assume page-level locking is used, task: find oldest male and female persons)

1. T1 locks all pages containing person records with sex=male, and finds oldest person (e.g., age=71)
2. T2 inserts a new male person with age=96
→ This record is inserted on a different page than the pages locked by T1
3. T2 deletes oldest female person with age=80
→ This record is also located on a page which is *not* locked by T1
4. T2 commits
5. T1 now locks all pages containing female person records and finds oldest (e.g., age=75)



There is no consistent DB state, where T1 is correct!

Index Locking

- T1 implicitly assumes that it has locked the set of all male person records
 - This is true only, if no records are added while T1 is executed.
 - Thus, some mechanism to enforce this assumption is needed.
- The example shows that conflict serializability is only guaranteed if the set of objects is fixed
- Possible solutions with locking on index level
 - No Index: T1 has to lock all pages and the file/table to prevent new records/pages being added
 - Index on sex field:
 - T1 needs to lock the index page with data entries for sex=male
 - If there are no such records yet, T1 must lock the index page where such a data entry would be created

Predicate Locking



Grant lock on all records that satisfy some logical predicate,
e.g., $\text{age} > 2 * \text{salary}$

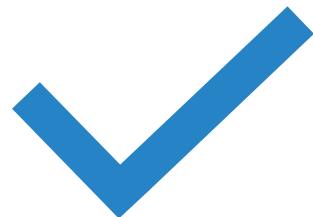


Index locking is a *special case* of predicate locking, for which an index supports efficient implementation of the predicate lock



In general, predicate locking has a lot of locking overhead, is expensive to implement and therefore not commonly used

Locking in B+-Trees



Simplistic strategy

Page level locking using some version of 2PL

Problem: very high lock contention in higher levels of the tree



Observations for a B+-Tree

Higher levels guide only the search, „real“ data is stored at the leaf level

For inserts, a node must be locked only if a split can propagate up to this node from a modified leaf node

Simple Tree Locking Algorithm

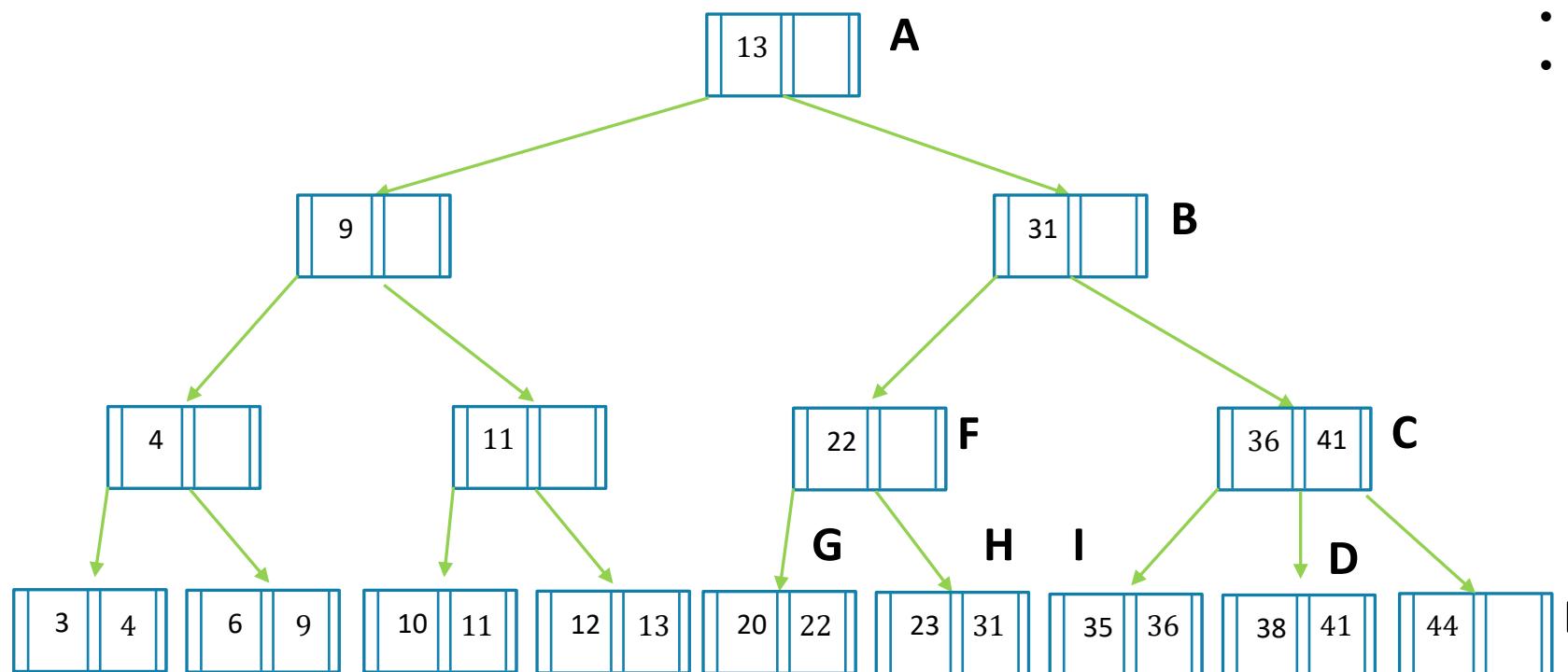
Search

- Start at root and go down
- Repeatedly, read lock child, then unlock parent

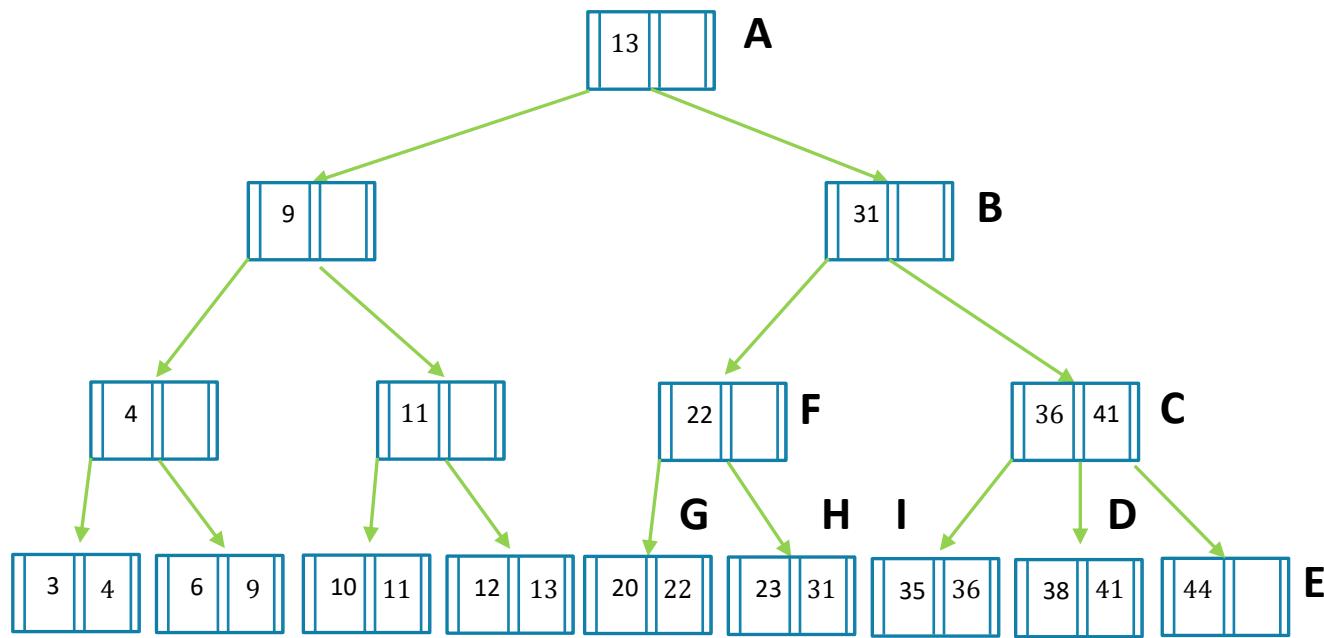
Insert/Delete

- Start at root and go down
- Repeatedly, write lock child, then check if it is *safe*
 - **Safe:** node has properties, such that changes will not propagate up beyond this node
 - Inserts: Node is not full
 - Deletes: Node is not half-empty
 - If node is safe, then unlock all its ancestors
- **Disadvantage:** Many write locks on nodes which are often not modified, as modification is only required in leaf node

Example

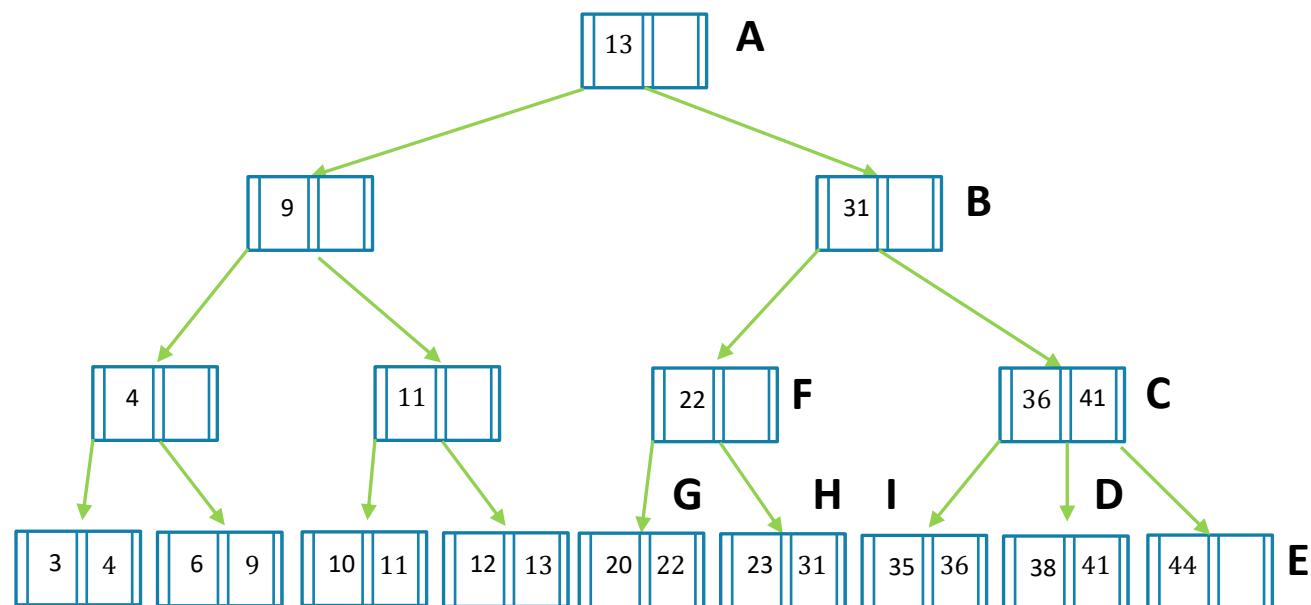


Example: Delete 38



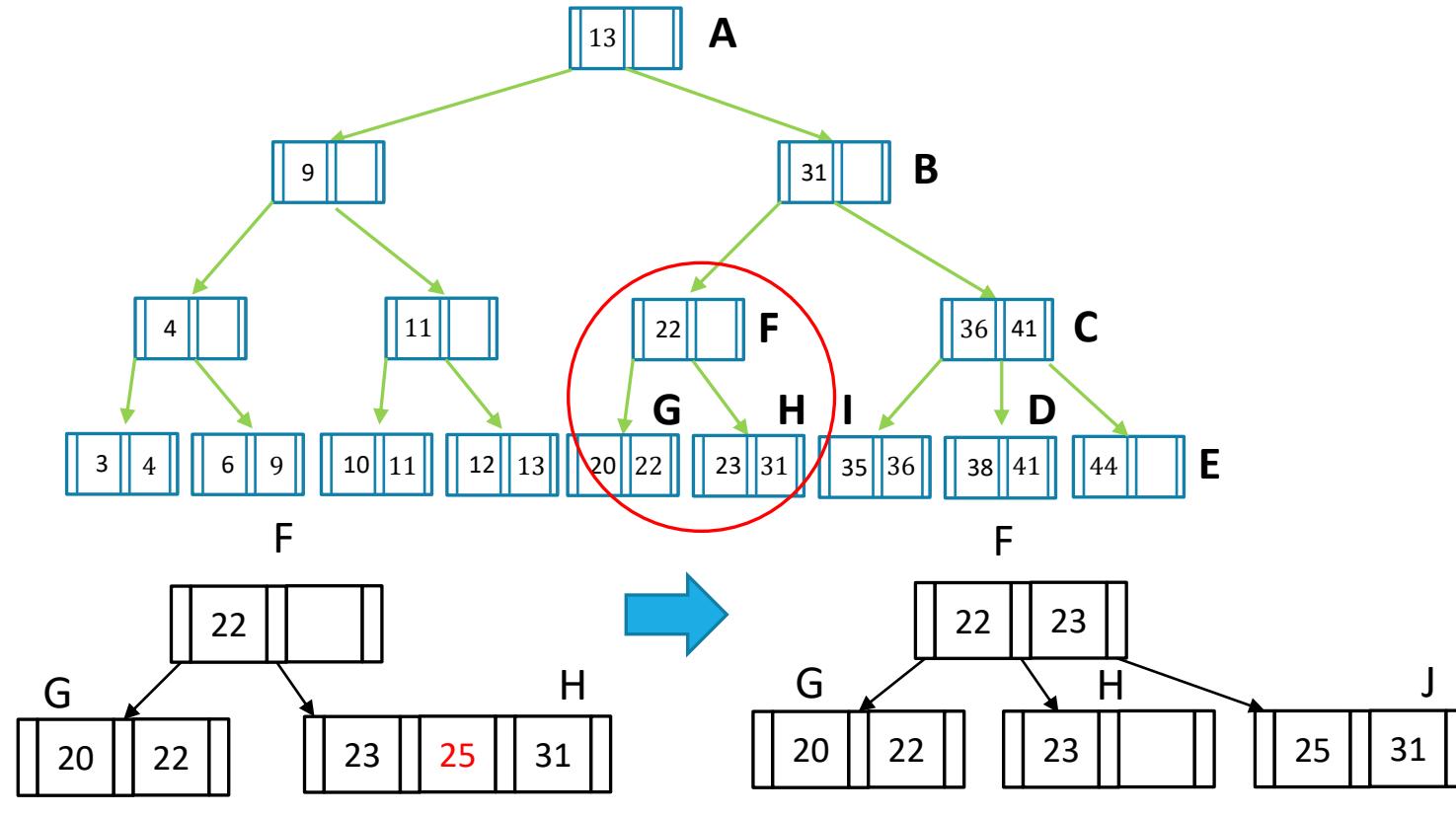
- $wl(A) r(A)$
- $wl(B) r(B) \rightarrow B$ is half-empty
 \rightarrow cannot unlock A
- $wl(C) r(C) \rightarrow C$ is not half-empty =safe
 \rightarrow can unlock A and B
- $wu(A) wu(B) \rightarrow$ go from the higher to the lower levels
- $wl(D) r(D) \rightarrow$ is not half empty = safe
 \rightarrow can unlock C
- $wu(c) w(D) wu(D)$

Example: Insert 45



- $wl(A) r(A)$
- $wl(B) r(B) \rightarrow B \text{ is not full} = \text{safe}$
 $\rightarrow \text{can unlock } A$
- $wu(A)$
- $wl(C) r(C) \rightarrow C \text{ is full} \rightarrow \text{cannot unlock } B$
- $wl(E) r(E) \rightarrow E \text{ is not full} = \text{safe}$
 $\rightarrow \text{can unlock } B, C$
- $wu(B), wu(C)$
- $w(E) wu(E)$

Example: Insert 25



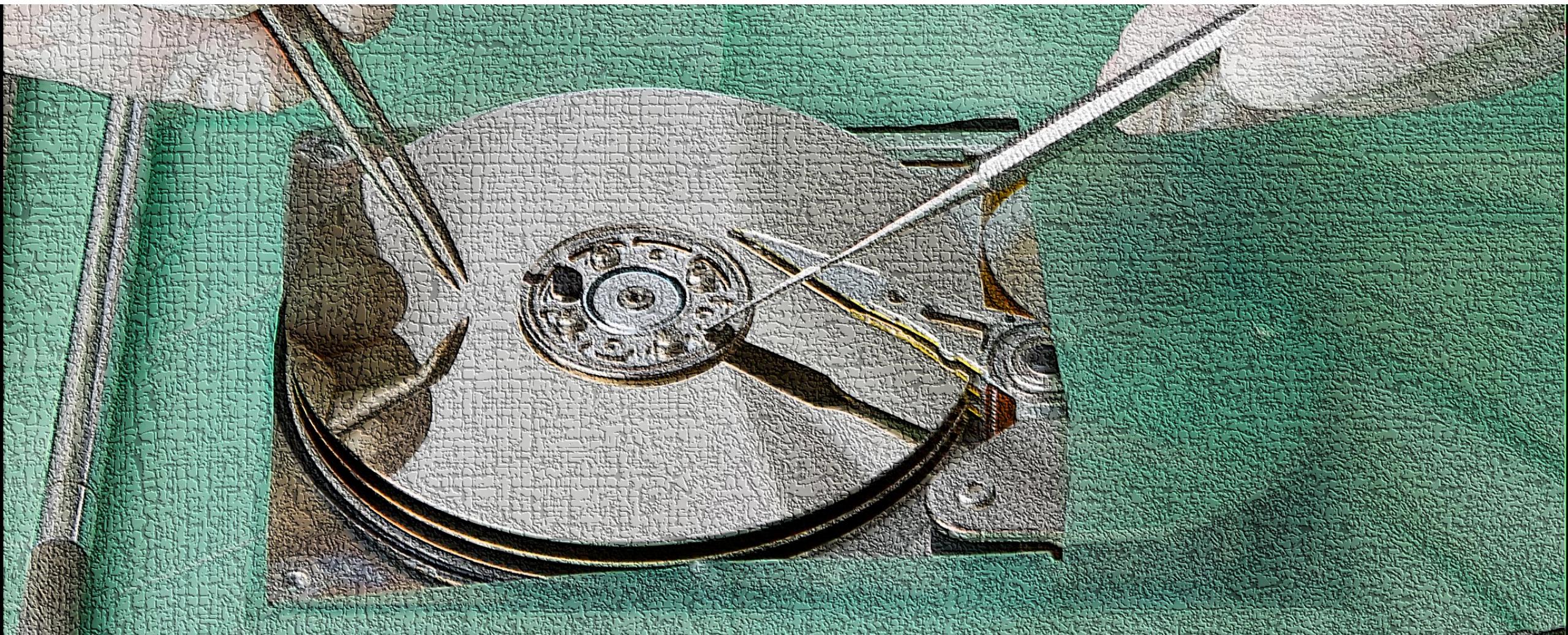
- $wl(A) r(A)$
- $wl(B) r(B) \rightarrow B$ is not full = safe
→ can unlock A
- $wu(A)$
- $wl(F) r(F) \rightarrow F$ is not full = safe
→ can unlock B
- $wu(B)$
- $wl(H) r(H) \rightarrow$ overflow
- $create(J)$ redistribute(H,J,F) $wl(J) w(J)$
 $w(H) w(F)$
- $wu(F) wu(H) wu(J)$

Improved Tree Locking

- Search: As before
- Insert/Delete
 - Set locks as for search, write lock only leaf
 - If leaf is not safe (i.e. updates propagate to parent), release all locks and restart the transaction using the previous algorithm
- Gambles that only leaf node will be modified
 - If not, read locks set on the first pass are wasteful
 - In practice, better than previous algorithm
- Alternative: Use intentional locks as in multigranularity locking (MGL)
 - Use iwl locks instead of write locks as in simple algorithm
 - Upgrade locks top-down from iwl to wl, modifications are propagated to parent nodes
 - Deadlocks possible

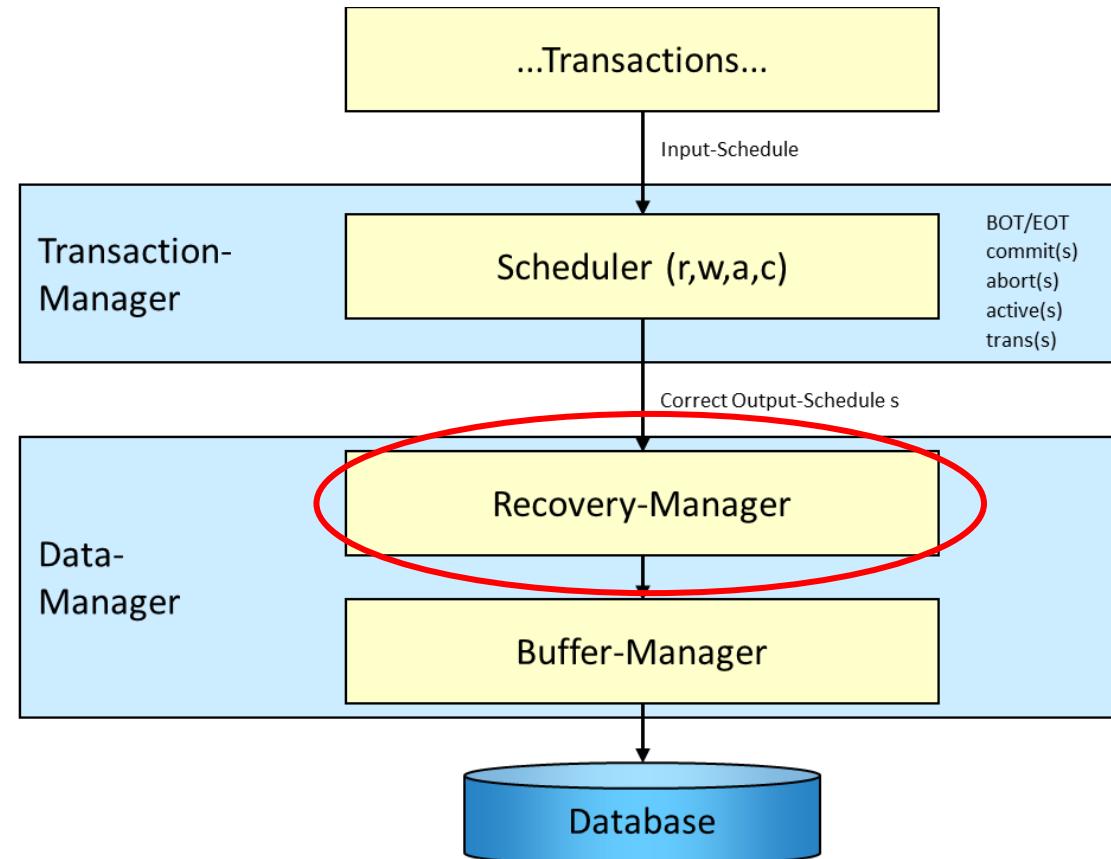
Concurrency Control without Locking

- Optimistic CC
 - Assumption: Conflicts are rare
 - Modifications are done on private copies of the data objects
 - Before the commit of a TX, check whether there is a conflict. If yes, abort TX and restart it. If not, private copies are copied into database.
- Timestamp-based CC
 - At startup, each TX t_i gets a timestamp $ts(t_i)$
 - If $p_i(x)$ and $q_j(x)$, $i \neq j$, are in conflict, then $p_i(x)$ is executed before $q_j(x) \Leftrightarrow ts(t_i) < ts(t_j)$
 - Generates conflict serializable schedules
 - Recoverability requires additional modifications (e.g., buffering of write actions), as dirty read problem is not avoided
 - Thus, Timestamp-based CC is unlikely to be more efficient than lock-based CC, but can be used in distributed database systems



6.7 Recovery Protocols

Transaction Processing Layers of a DBMS



Learning Goals

At the end of this section you will be able to

- ✓ explain reasons for UNDO and REDO actions
- ✓ explain the REDO/UNDO protocol
- ✓ explain and apply the ARIES protocol



Types of faults, DBMS must be able to handle

1. Transaction faults

A transaction does not reach its commit point, e.g., by an error in program or an involvement in a deadlock.

2. System crash

Parts of (volatile) main memory or buffers get lost, e.g., by errors in DBMS code, in operating systems, or hardware.

3. Media fault

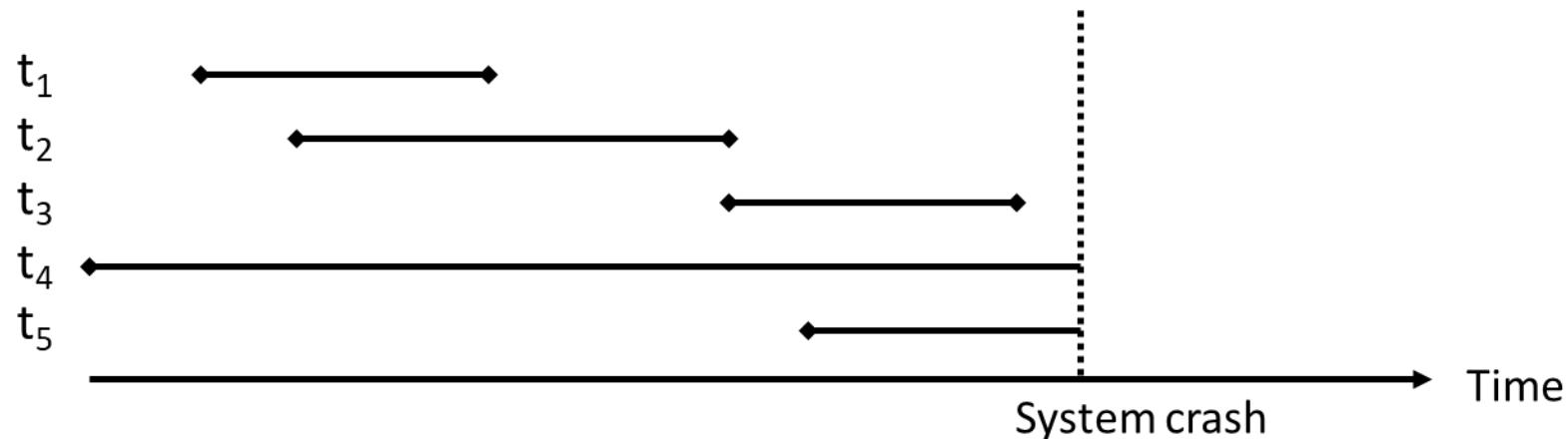
Parts of (non-volatile) secondary storage get lost, e.g., by a head crash on a disk, faults in an operating system routine for the writes onto disks.

In the following only fault types (1) and (2) will be considered.

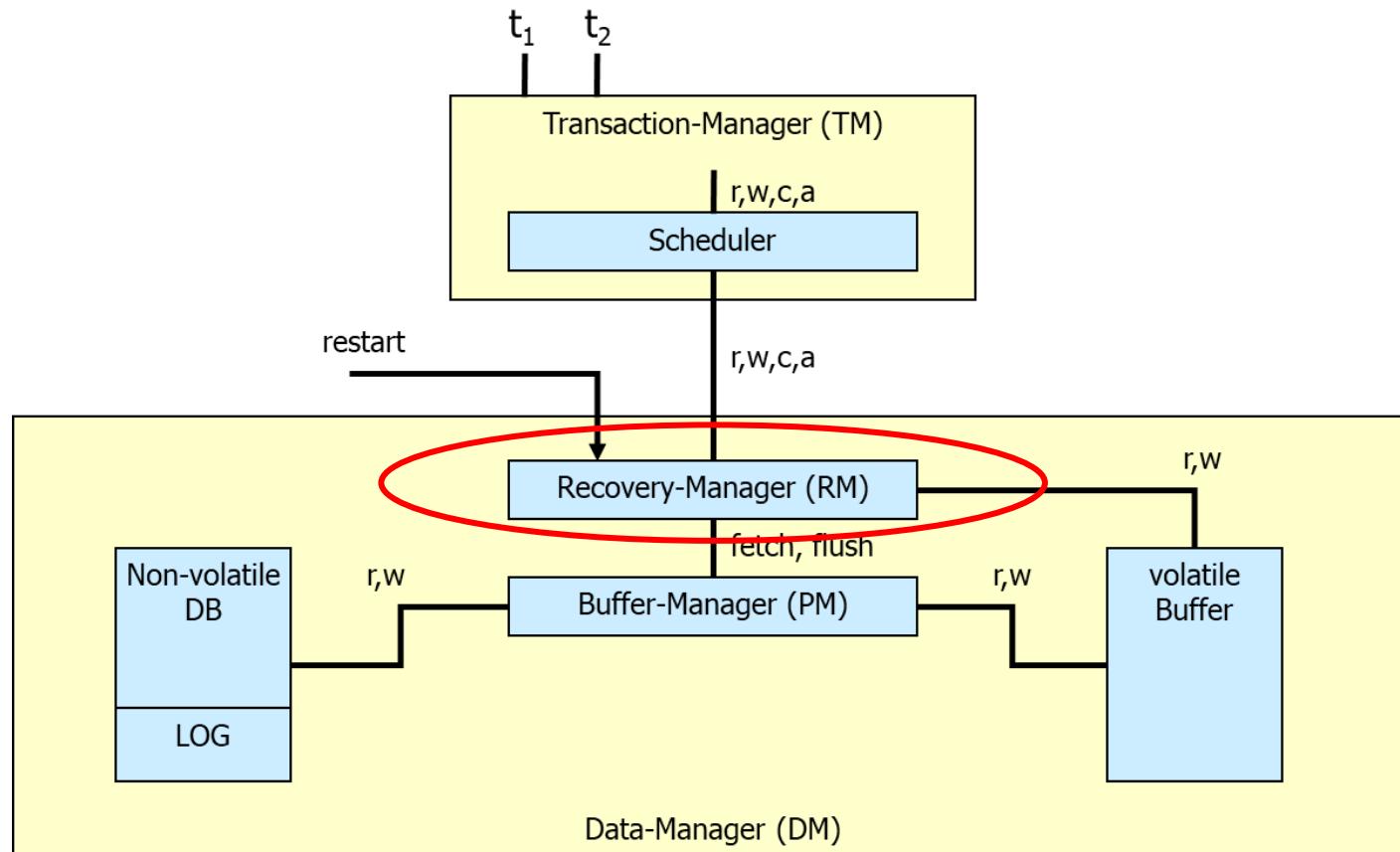
Crash Scenario

We can classify TXs in two classes

1. TXs already **released** before the fault.
These need a **REDO**, if results have not been permanently stored.
2. TXs **still active** by the time of the fault.
These need an **UNDO**, if some results are already stored permanently on disk.



Organization of a Data Manager



Recovery Manager

The Recovery Manager (RM) maintains a **log file**, which is used as follows:

- If TX wants to write a new value of x, a ***Before-Image*** of x is written in the log beforehand
(consists TX ID, ID of x, and old value of x)
- New value of x is logged in an ***After-Image***
(consists of TX ID, ID of x, and new value of x)
- Execute REDO or UNDO of TX: log entries for TX are read and processed in ***reverse sequence***

Classification of recovery protocols:

- How is the log structured and how is it maintained?
(Are only Before- or only After-Images are stored or both?)
- When are the newly written pages propagated from the buffers into the DB? (avoid hot spots)

Assumptions

- Read- or write-operations refer to a page of secondary storage.
- **Read operation:** transfers a page from the DB into the buffer, if the corresponding page is not yet in the buffer.
- **Write operation:** modifies content of a page (must be in buffer); page can be written to the database at once or in a later time.

REDO and UNDO

REDO

- Required by RM, if TXs are allowed to **terminate, before** all written values are persisted in DB
 - If a system fault occurs, the effects of already committed TXs may be **missing** in the database
- These transactions should be repeated by REDO.

UNDO

- Required by RM, if TXs that are **still running** are allowed to **write** to the DB
 - If a system fault occurs, the database may **contain values not committed** yet
- the concerning transactions must be reversed by UNDO

UNDO and REDO rules

Each protocol must satisfy the following UNDO and REDO rules

- **UNDO-rule („Write-Ahead-Log-Protocol“)**

The *Before-Image* of a write operation (old value of x) must be written into the log, *before* the new value appears in the stable database.

- **REDO-rule („Commit-rule“)**

Before a TX is terminated, every new value that has been written by it must be in the stable storage (in the stable database or in log).

Direct consequence

- **For No-REDO:** ensure, that all After-Images of a transaction are written in the DB before or during commit.
- **For No-UNDO:** ensure, that no After-Image of a tx is written into the DB (but only the log) before commit.

UNDO/REDO Protocol

Idea: The RM does not control the buffer manager with respect to the writing of pages into the database!

→ It depends on the page replacement strategies of operating systems, when to execute a flush operation of buffers, e.g., by Demand Paging:

- a write operation writes in (log and) buffer page p
- if p is replaced (written into the database) and the concerning TX aborts, UNDO is necessary
- If a TX reaches its commit and p is damaged by a crash before a flush, REDO is necessary

Alternatives to UNDO/REDO Protocol

Steal

- An object o changed by TX t is written to disk before t commits (**dirty page**).
- Buffer manager might replace a frame in memory which contains page with object o (i.e., frame is stolen from t)
- If this allowed, then a **steal** approach is used.

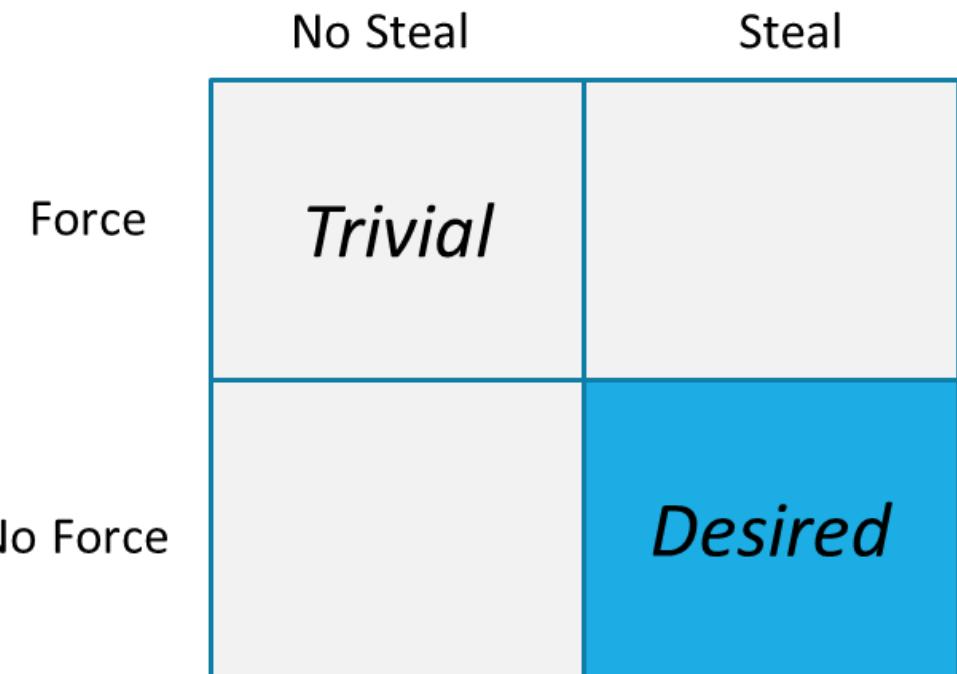
Force

- When a TX commits, it is ensured that all changes it has made to objects in buffer are immediately **forced** to disk

Steal and Force – Pros and Cons

- Force every write to disk?
 - Poor response time
 - But durability is guaranteed

- Steal buffer frames from uncommitted transactions?
 - If not, poor throughput
 - If yes, how can we ensure atomicity?



More on Steal and Force

Steal Approach (why enforcing Atomicity is hard)

- A page p is written to disk while a TX still holds a lock
- What happens if the TX with the lock on p aborts?

→ Old value of p at steal time must be remembered, so that UNDO is possible

No-Force Approach (why enforcing Durability is hard)

- What if the system crashes before a modified page is written to disk?
- Write as little as possible at commit time, in a convenient place, to support REDO of modifications

Example: ARIES - Steal, No-Force-Approach

Three phases for the RM after a crash:

1. Analysis

- Identify ***dirty pages*** in the buffer and update Dirty Page Table (DPT)
- Identify ***active TXs*** at the time of the crash and update Transaction Table (TT)

2. Redo

- Repeat all actions from the log, starting from the first action which made a page dirty
- Restores the database state to what it was at the time of the crash

3. Undo

- Undo TXs that did not commit, so that DB reflects only committed transactions

Example: ARIES - Three main principles

1. Write-Ahead-Logging

2. Repeating history during Redo

- Repeat *ALL* actions of the DBMS before a crash, restoring the exact state at the time of the crash

3. Logging changes during Undo

- Changes made to the database while undoing a transaction are logged to ensure such an action is not repeated in the event of repeated failures/restarts
- This information is written into the log in **Compensation Log Records (CLRs)**

Write-Ahead Logging (WAL)

1. Must force the log record for an update before the corresponding data page gets to disk
 2. Must write all log records for a TX *before commit*

1. guarantees Atomicity,
 2. guarantees Durability

ARIES: Structure of the Log

- Fields of a log record:
 - LSN (Log Sequence Number, ID for a Log record)
 - Transaction ID
 - Type (Update, Commit, Abort, End (for End of Commit/Abort), CLR)
 - pageID
 - Offset
 - Length
 - old data
 - new data
- Each data page has a *pageLSN*: The LSN of the most recent log record with an update for that page
- System maintains *flushedLSN*: The max LSN flushed so far

Checkpointing

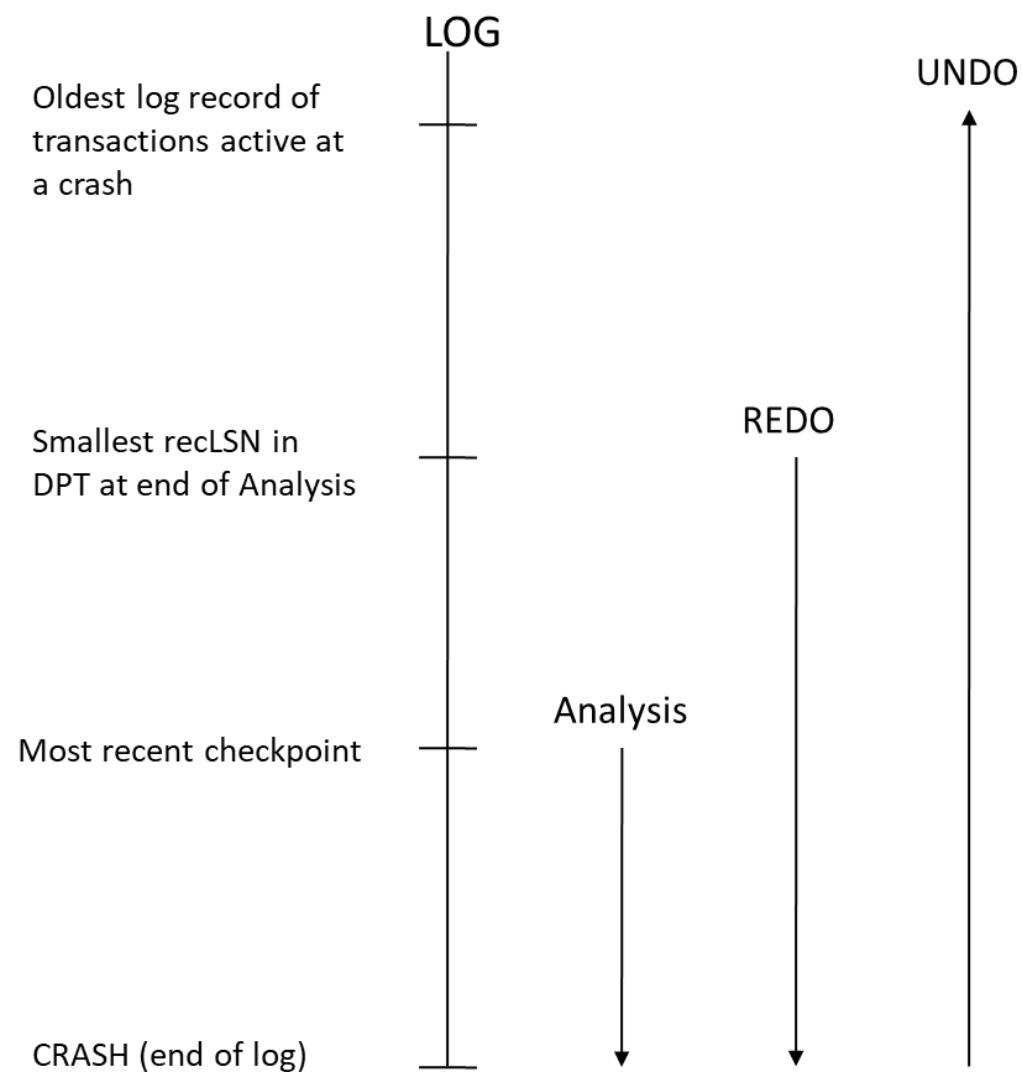
- Periodically, DBMS creates a checkpoint, to minimize time taken to recover from system crashes

- Write to the log:
 1. Begin checkpoint record: Indicates the start of checkpointing
 2. End checkpoint record: current transaction table and Dirty Page Table (**DPT**)
(as at the time of „begin checkpoint“ record is written)
 3. Master record: store LSN of the „begin checkpoint“ record in a safe place in stable storage

This is called a „*Fuzzy Checkpoint*“

- Inexpensive, because not all pages in the buffer have to be written to disk
- But limited by oldest unwritten change in a dirty page → flushing dirty pages periodically is a good idea
- Other transactions can run concurrently to checkpointing

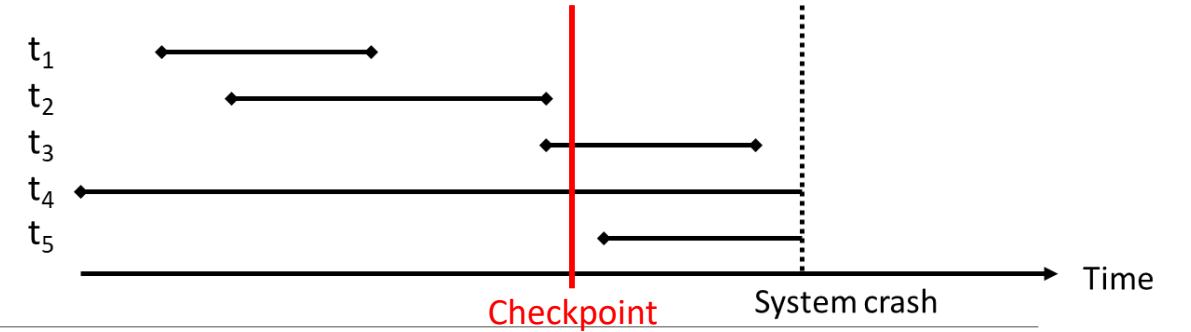
Crash Recovery



Example

Log

LSN	LAST_LSN	TX_ID	TYPE	PAGE_ID
1	0	T1	update	123
2	0	T4	update	234
3	1	T1	commit	234
4	0	T2	update	35
5	4	T2	commit	35
6	0	T3	update	234
7	Begin CKPT			
8	End CKPT			
9	0	T5	update	123
10	6	T3	commit	234



TT

TX_ID	LAST_LSN	STATUS
T1	3	commit
T2	5	commit
T3	6	running
T4	2	running

LSN of last update of TX

TT contains all TXs which are active (i.e., running, abort, commit)

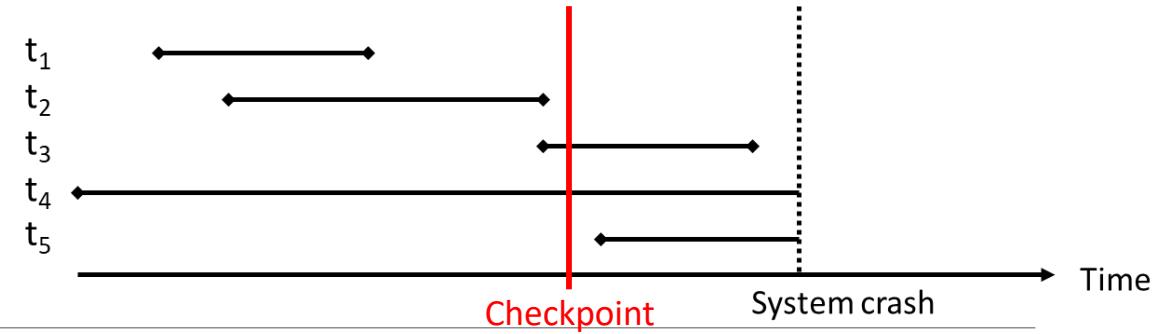
DPT

Page_ID	LSN
35	4
123	1
234	2

Dirty pages in buffer.

First LSN which made this page dirty

Example – Analysis



Log

LSN	LAST_LSN	TX_ID	TYPE	PAGE_ID
1	0	T1	update	123
2	0	T4	update	234
3	1	T1	commit	234
4	0	T2	update	35
5	4	T2	commit	35
6	0	T3	update	234
7	Begin CKPT			
8	End CKPT			
9	0	T5	update	123
10	6	T3	commit	234

TT

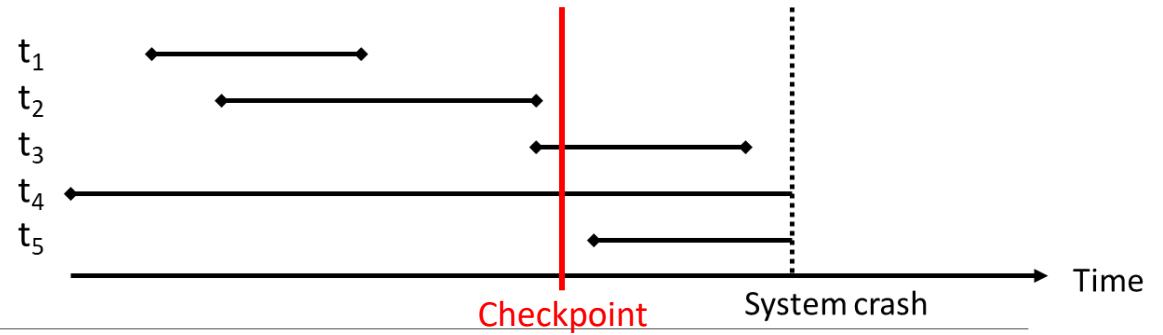
TX_ID	LAST_LSN	STATUS
T1	1	commit
T2	5	commit
T3	6	commit
T4	2	running
T5	9	running

DPT

Page_ID	LSN
35	4
123	1
234	2

no updates here

Example – REDO



Log

LSN	LAST_LSN	TX_ID	TYPE	PAGE_ID
1	0	T1	update	123
2	0	T4	update	234
3	1	T1	commit	234
4	0	T2	update	35
5	4	T2	commit	35
6	0	T3	update	234
7	Begin CKPT			
8	End CKPT			
9	0	T5	update	123
10	6	T3	commit	234

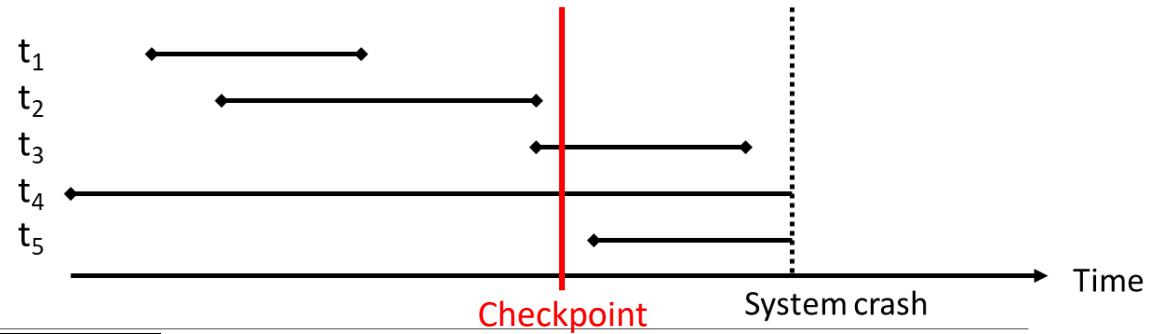
DPT

Page_ID	LSN
35	4
123	1
234	2

REDO from smallest LSN in the DPT (LSN = 1) until the time of the crash

REDO all actions (even of aborted TXs, CLR)

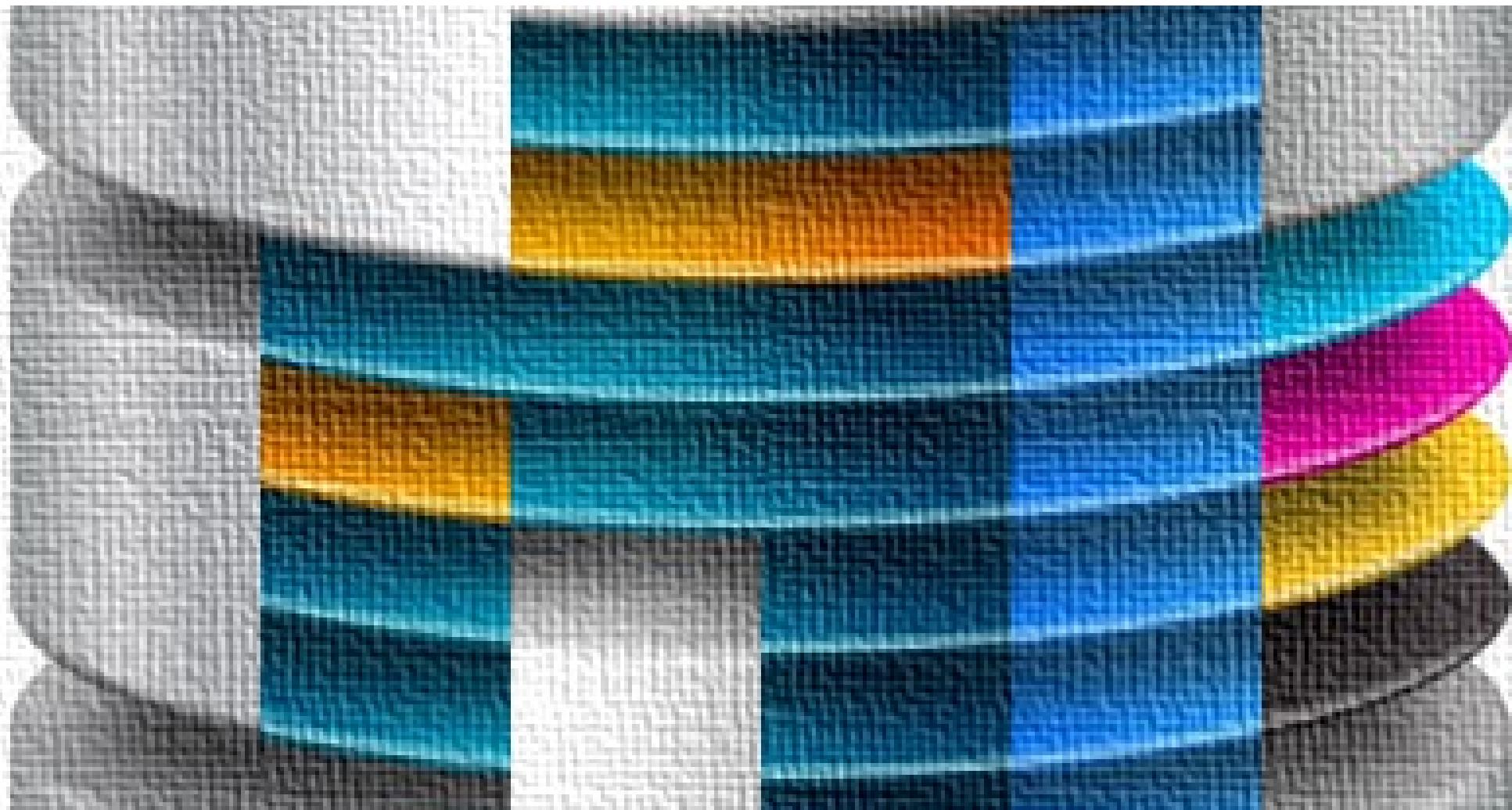
Example – UNDO



Log	LSN	LAST_LSN	TX_ID	TYPE	PAGE_ID	undonextLSN
	1	0	T1	update	123	
	2	0	T4	update	234	
	3	1	T1	commit	234	
	4	0	T2	update	35	
	5	4	T2	commit	35	
	6	0	T3	update	234	
	7	Begin CKPT				
	8	End CKPT				
	9	0	T5	update	123	
	10	6	T3	commit	234	
	11	9	T5	CLR	234	2
	12	2	T4	CLR	123	0

TT	TX_ID	LAST_LSN	STATUS
	T1	1	commit
	T2	5	commit
	T3	6	commit
	T4	2	running
	T5	9	running

UNDO for running TXs in reverse sequence until oldest LSN of TXs from the time of the crash



6.8 Concurrency Control in SQL

Review of Transaction Theory

ACID Principle

- Atomicity : all or nothing
- Consistency : from one consistent state to next
- Isolation : other users can be ignored
- Durability : committed transaction results are safe

Conflict serializability of committed transactions

- prevents lost updates and inconsistent reads
- is testable at any time
- with polynomial effort (cycle-free conflict graph)

Strictness of schedules (with aborted/running transactions)

- allows prevention / recovery of dirty reads (recoverability)
- avoids cascading aborts
- simplifies recovery process

Review of Scheduling & Recovery Algorithms

Strict two-phase locking (S2PL)

- **guarantees conflict serializability** without cycle testing
- **guarantees strictness** and thus quick recovery processes
- **can be extended** to multiple granularity locking (MGL) and tree structures (e.g., B tree locking)

UNDO/REDO recovery protocols

- enable **independence** of buffer management and transaction management
- Write-Ahead Log writing of old values **guarantees atomicity**
- Log writing of new values before commit **guarantees durability**
- can even **deal with repeated crashes** during the recovery process itself
- can be optimized for performance by (fuzzy) checkpoints

SQL Isolation Levels

- Isolation level of transactions can be controlled by a SQL option
- Define the type of locks acquired on read operations

- Useful to tune database application

```
SET TRANSACTION ISOLATION LEVEL
{
    READ COMMITTED
    | READ UNCOMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
}
```

- Isolation level can also be set in each SELECT statement:

```
SELECT * FROM Employee WITH READ COMMITTED
```

SQL Isolation Level Descriptions

- READ UNCOMMITTED
 - sets no locks at all => reads also uncommitted data => dirty read possible
 - not possible in Oracle
- READ COMMITTED
 - reads only committed data => dirty reads are avoided, but non-repeatable reads/phantom reads are possible
 - default in MS SQL Server, Oracle
- REPEATABLE READ
 - locks are placed on all data that is used in a query, prevents updates on this data (avoids non-repeatable read problem)
 - phantom problem still possible (insertions and deletions on the data set which is used in the query)
 - not possible in Oracle
- SNAPSHOT (only SQL Server)
 - data read by a tx will be the transactionally consistent version of the data that existed at the start of the transaction
→ sees only modifications committed before start of the tx
 - No blocking of other writing txs, writing txs do not block reading snapshot txs
- SERIALIZABLE
 - uses range lock on a data set, which is used in a query
 - lowest concurrency level
 - In Oracle this is equal to Snapshot Isolation

See: <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide>

Which Problems are still possible in which level?

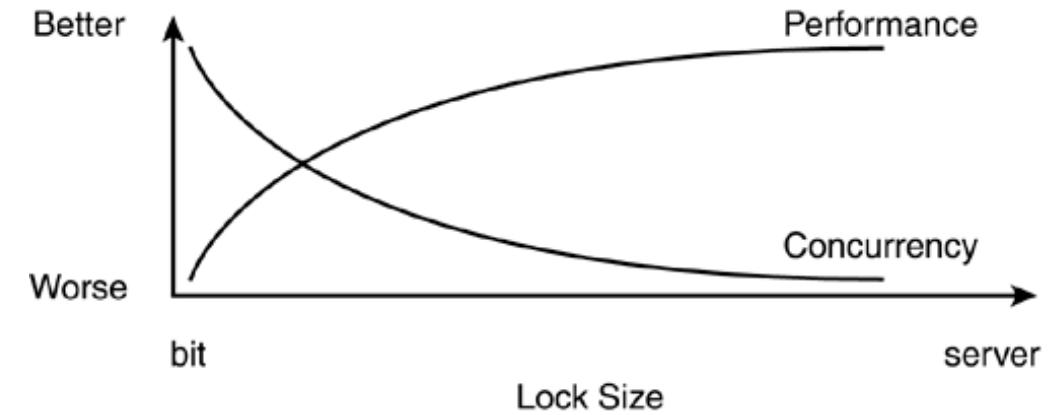
	Dirty Reads	Lost Update	Non-repeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

Lock Granularity in MS SQL Server

Locking Hints in MS SQL Server

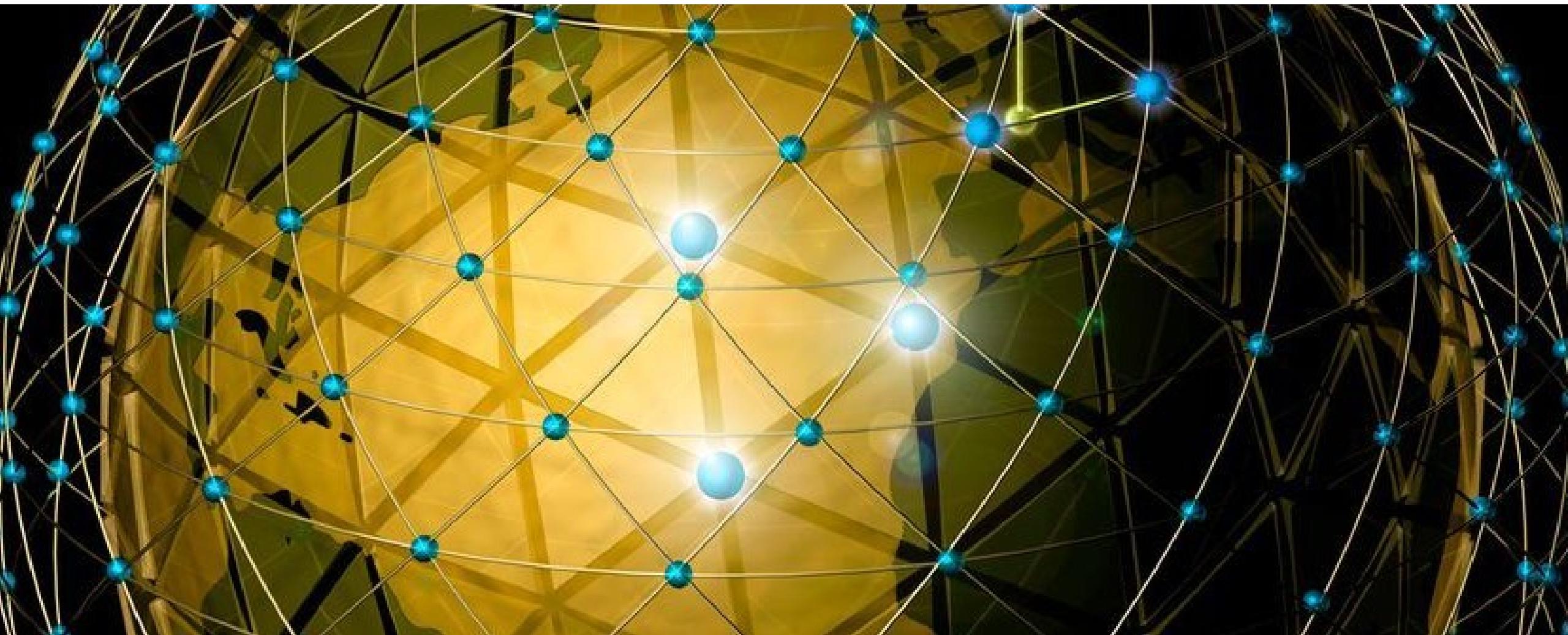
SELECT ... FROM <table> WITH <locking_hint>

- NOLOCK (equivalent to READ UNCOMMITTED)
- READCOMMITTEDLOCK (read committed via locking)
- ROWLOCK
- PAGLOCK
- TABLOCK
- TABLOCKX: hold table lock until end of transaction
- HOLDLOCK: hold lock until end of transaction
- XLOCK: exclusive lock (in combination with PAGLOCK or TABLOCK)
- UPDLOCK: use update locks instead of shared locks
- READPAST: skip locked rows



Lock Granularity in Oracle

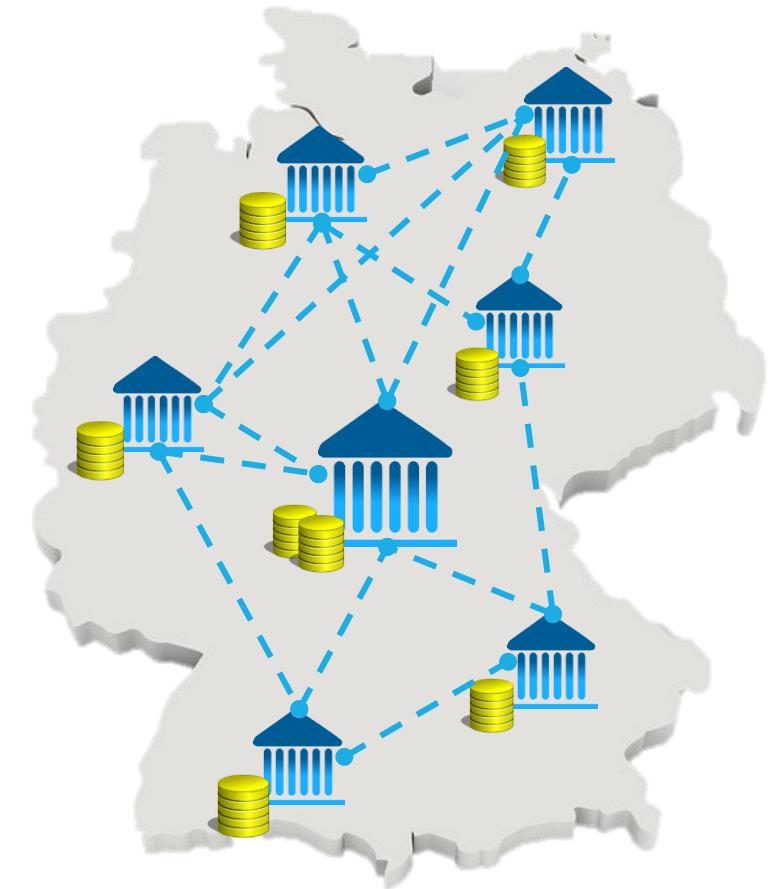
- Automatic acquisition of table and row locks
- LOCK TABLE command
 - LOCK TABLE <table> IN <lockmode>
 - ROW SHARE/SHARE UPDATE
 - ROW EXCLUSIVE
 - SHARE
 - EXCLUSIVE
- FOR UPDATE OF ...
 - SELECT ... FROM <table1>, <table2>, WHERE ... FOR UPDATE [OF <table2>,....]
 - Locks rows and tables in a query for update operations
 - If “OF” is not used, all tables in FROM part will be locked
 - Related: FOR UPDATE in DECLARE CURSOR statement (Embedded SQL)



6.9 Transaction Management in Distributed Systems

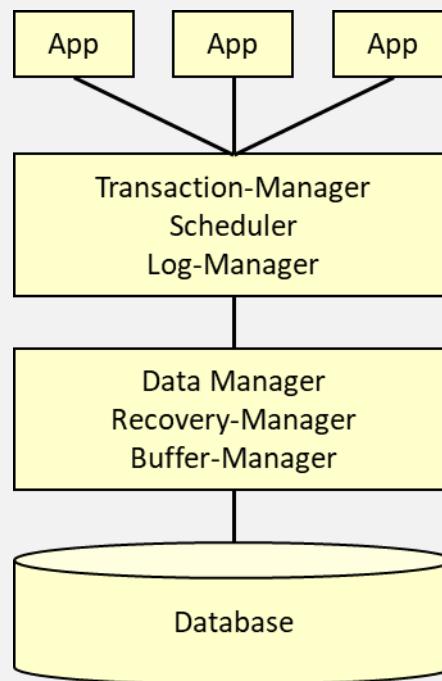
Distributed Databases – Revisited

- Examples: Bank branches with accounts of local customers, logistics, merchandise management, Gmail, Adwords
- Several computer nodes connected via communication network
- Node: Instance of DBMS + part of the database
- Partitioned: Fragmentation + Allocation
- Replicated: additional replication of fragments
- Nodes should work as autonomously as possible

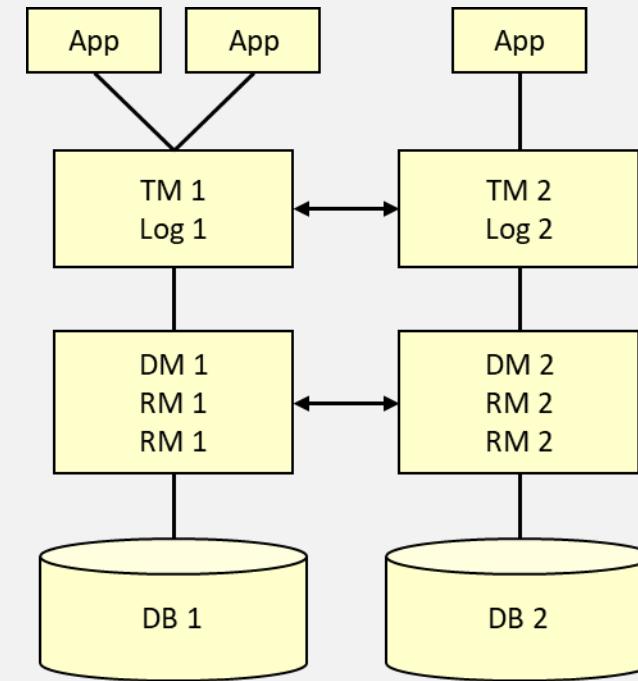


Differences in Architectures

So far: Applications compete for one resource



Now: More applications compete for several resources



RM: Resource Manager,
identified by RMID;
provides resource incl.

Log data
(commit/abort)

CM: Communication
Manager, accepts
„transactional remote
procedure calls“ (TRPC)
from applications or
other TMs (through its
CMs)

Problems for Concurrency Control

Multiple copies of data items

maintain consistency
between them especially
after failure and
recovery of a site

Failure of individual nodes

system must continue to
operate without them

Failure of communication

network partitioning, only
nodes in the same partition
can communicate

Distributed commit

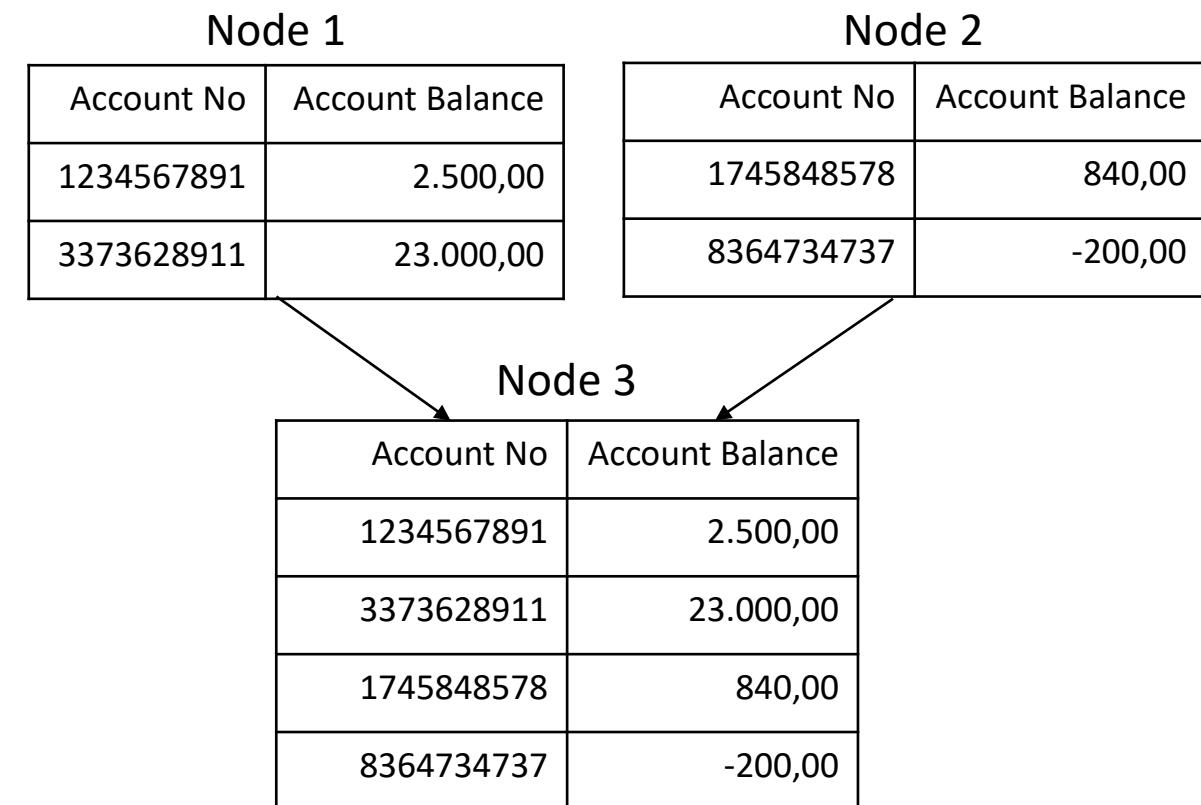
How to deal with distributed
transactions (if participating
nodes fail)?

Distributed deadlock

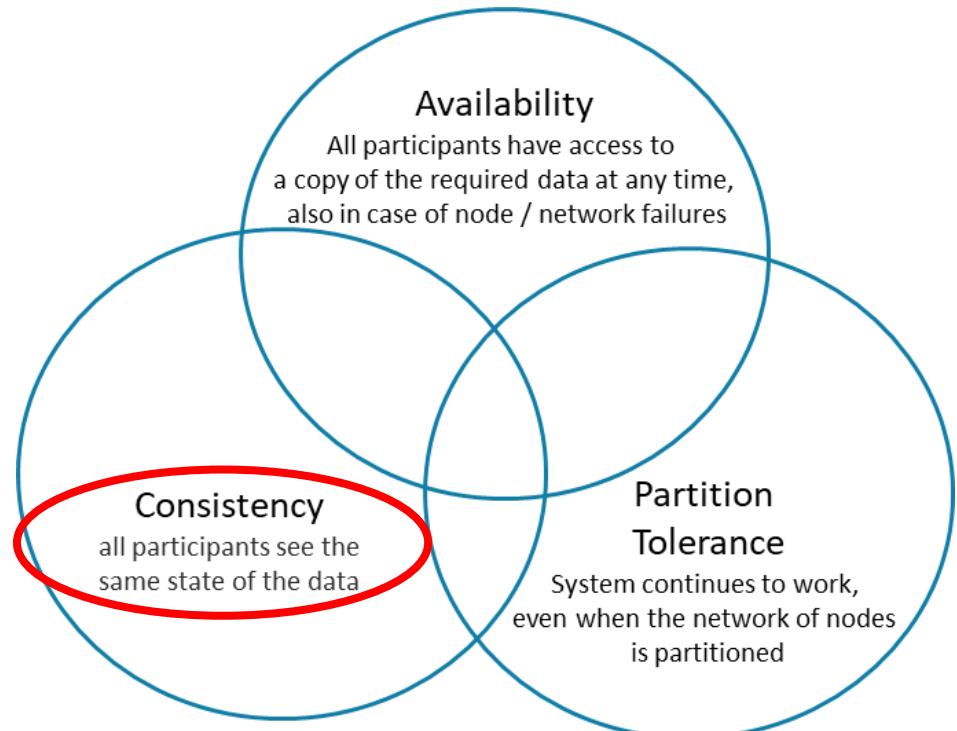
deadlock can occur among
multiple nodes

Consistency in Distributed Databases

- Single node: as seen
- **Multiple nodes:** *Distributed TX execution*, changes of a TX must be made in all affected nodes
- **With replicas**
 - **Simple:** All replicas of a data object are identical
 - **Strict:** all records of all replicas must be the same



ReCAP



- **Strong consistency**

Changes are immediately communicated to all nodes

- **Weak consistency**

Cannot guarantee that changes have been communicated

- **Eventual Consistency**

Guarantees that nodes will eventually receive the changes

ACID & CAP

Atomicity	Consistency	Isolation	Durability
<ul style="list-style-type: none">• If focus is on availability, both sides should still use atomic operations• Higher-level atomic operations simplify recovery	<ul style="list-style-type: none">• In ACID it means transaction does not violate database rules• In CAP: refers only to single-copy consistency → in case of partitions need recovery to restore ACID consistency	<ul style="list-style-type: none">• During a partition, system can only operate on at most one side → serializability not working here	<ul style="list-style-type: none">• Not so easy, maybe durable operations need to be rolled back, if they unknowingly violated an invariant during partitioning

→ Having ACID TXs on all sides of partitioning makes recovery later on easier

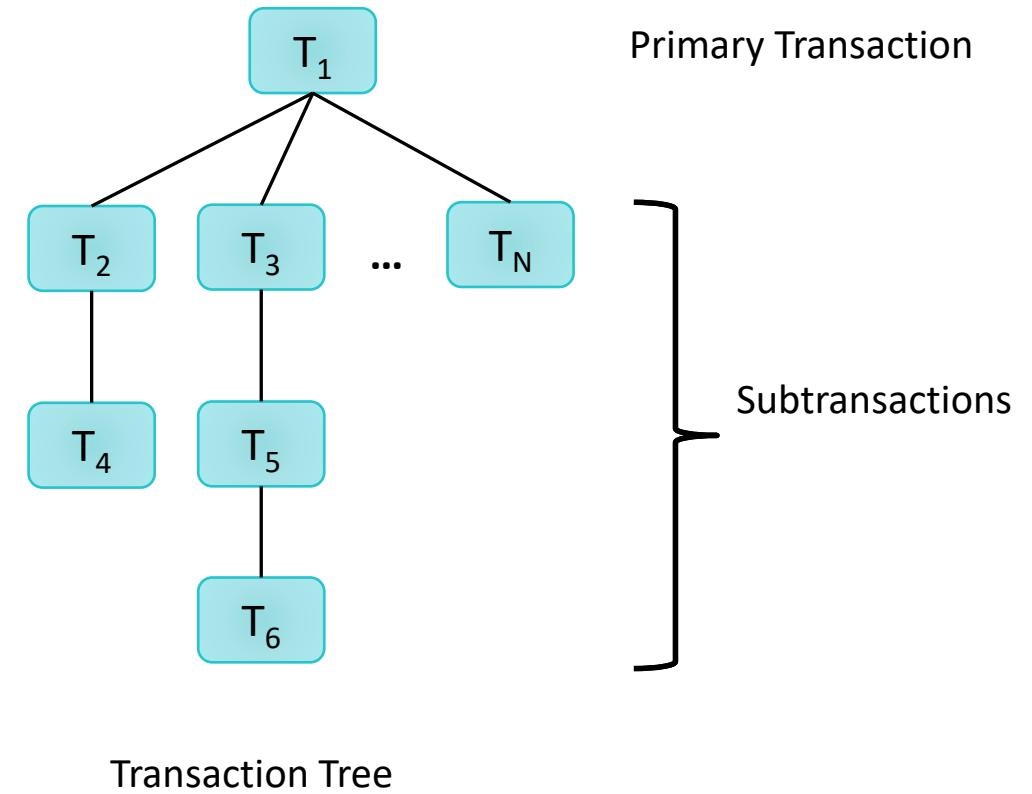
How to Achieve the ACID Principle in DDBs?

- How do I ensure that the changes are accepted or rejected in all nodes?
How is the recovery after a failure organized?
 - Commit protocols with DO, UNDO, REDO
- How to achieve multi-user operation in a distributed DBMS?
 - Locking protocols, timestamp procedures, optimistic synchronization,
Multi-version Concurrency Control (MVCC)
- How I can ensure consistency for replicas
 - Quorum consensus method, snapshot replication, write-all replication

Distributed TX Management

Goal: Changes are executed everywhere or not at all

- What shall happen if a single node cannot commit?
- What happens if one or more nodes fail / suffer network partitioning?
- As no real concurrency is achieved, how can I synchronize changes correctly?

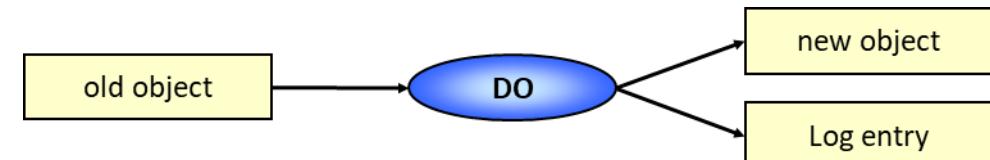


DO-UNDO-REDO Protocol

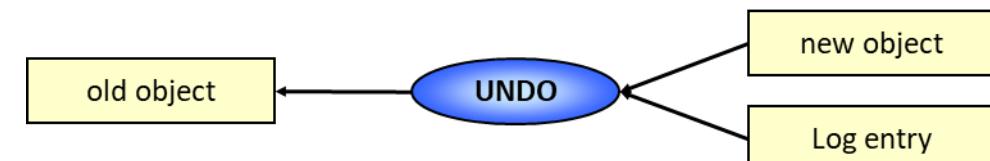
Requirement: Each RM must satisfy the DO-UNDO-REDO protocol

Write-ahead log: entry log for each node must be written before the DB fragment is actually updated

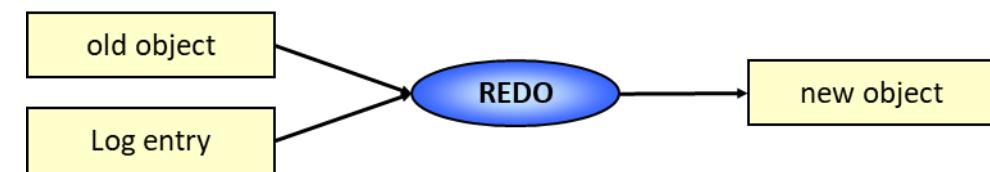
- **DO:** Execute operations and record them in the log file.



- **UNDO:** Reconstruct the old object state from the new object state and the respective entry in the log file.



- **REDO:** Reconstruct the new object state from the old object state and the entry in the log file (e.g., by renewed invocations of the operations, which stand in the log entry).



Distributed Commit & Concensus Protocols

- **Goal:** Changes are executed everywhere or not at all
- **Concensus:** All nodes agree on a decision (commit or abort?)
- Correctness criteria
 - Accordance: all participants end up with the same decision
 - Termination: all processes come to a decision
 - Validity: the concensus has been made according to the decision rules
- Further requirements: Robustness and maximum autonomy





Rules of the Game

1. Each node comes to a decision
2. Decision can not be reversed
3. Commit decision: only if **all** have decided for commit
4. No mistakes, all decide for commit
→ Commit
5. Protocol terminates

Reasons to Refuse the Commit Request

- A participating user confirms a CANCEL-operation, which is at first stored in a buffer of a RM.
- A RM collects during a tx certain parts of an object (e.g. email with sender, receiver, text). If by a commit request not all the parts are presented, the RM will disagree.
- A RM performs integrity tests, which can be tested only at the end of a transaction (e.g. $x.spouse=y \Rightarrow y.spouse=x$). If an integrity constraint is violated, thus disagree.
- The RM performs in the end certain clean up operations, for which it requires locks, which are not granted.

Commit Operation – 4 Steps

- Prepare step

- Coordinator sends for a committing transaction TID a **prepare message** to all the concerning RMs, i.e., it asks for a voting over the commit of TID.

- Decide step

- If all have agreed, the **coordinator writes commit(TID) into the log**. The transaction then has executed a „logical commit“.

- Commit step

- Coordinator informs each concerning TM about the commit decision. **Each node then executes its local commit** for the TID.

- Completion step

- If all the concerning TM have confirmed the commit step, a **commit_complete(TID)** will be written into the log. The transaction TID is then terminated.

Eager commit:

The write of commit records on the log is forced to be an atomic operation.

Lazy commit:

The write of log files is an asynchronous process. If a system crash occurs before the write of log records, thus a „logical committed“ transaction will still abort (violation of „durability“ in ACID)

Abort Operation – Also 4 Steps

- UNDO step
 - Coordinator **reads the log file backwards** and invokes UNDO operations (the concerning RM is invoked).
- Broadcast step
 - If all operations have been performed (or reached a „savepoint“), coordinator informs all concerning RMs, that the transaction TID was called off.
- Abort step
 - Coordinator writes **aborted(TID)** in the log.
- Completion step
 - Writes **abort_completed(TID)** in the log.

Note: The coordinator needs not to wait for the confirmation of the broadcast step of all RMs, because the transaction was already physically canceled after the UNDO step. That is: No matter how the RMs react to the broadcast step, the transaction TID is already by all means considered as „aborted“.

2-Phase-Commit-Protocol (2PC) (not related to 2PL!)

Also: centralized 2PC

Coordinator: dispatches tasks

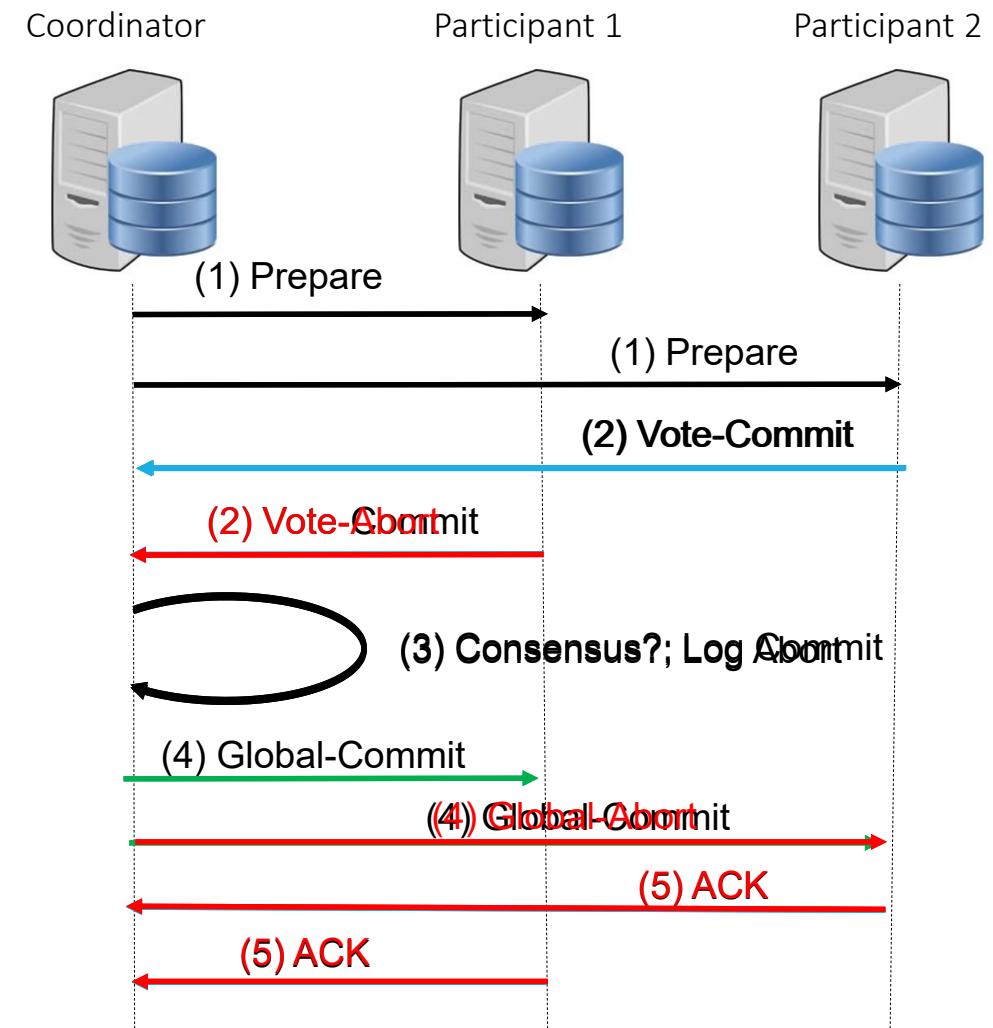
Participants: vote

Prepare phase

- (1) Coordinator asks participants if they agree to commit
- (2) Participants send decision to coordinator, local and central log entries

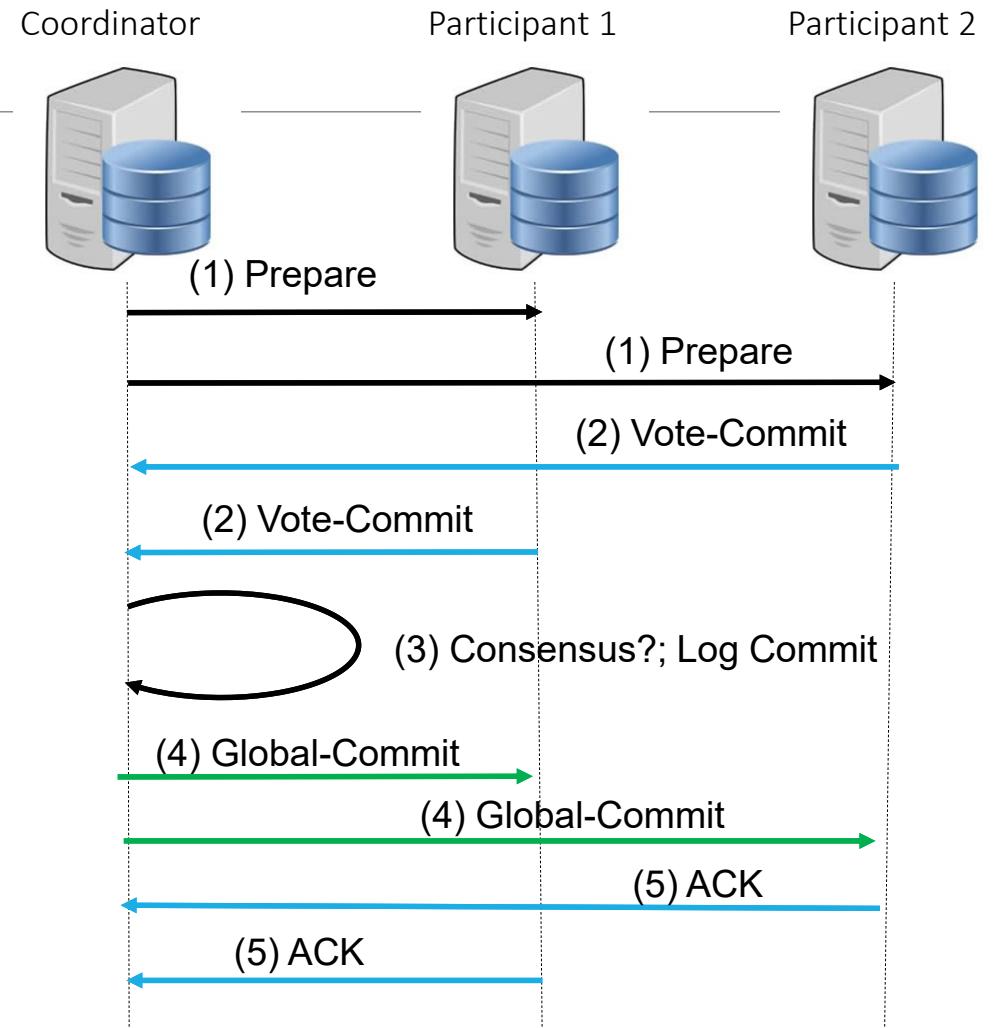
Commit phase

- (3) Coordinator makes decision based on votes, log entry
- (4) Participants with commit → wait for decision & commit
- (5) Participant acknowledges decision (ACK)



Problems and Recovery

- Failure of coordinator
 - After (2): As participants cannot take decision back → block these participants; all others who not decided yet: abort
 - After (3): Go on with protocol
- Failure of participant
 - Analyze log: failure before (2) → abort
 - After (2): ask coordinator for status
 - Global commit: REDO
 - Global abort: UNDO
 - After (4): REDO without asking
- Network Failure
 - Timeout → abort
- Failure of entire system
 - Try to create local log files from global log file



Linear 2PC Protocol

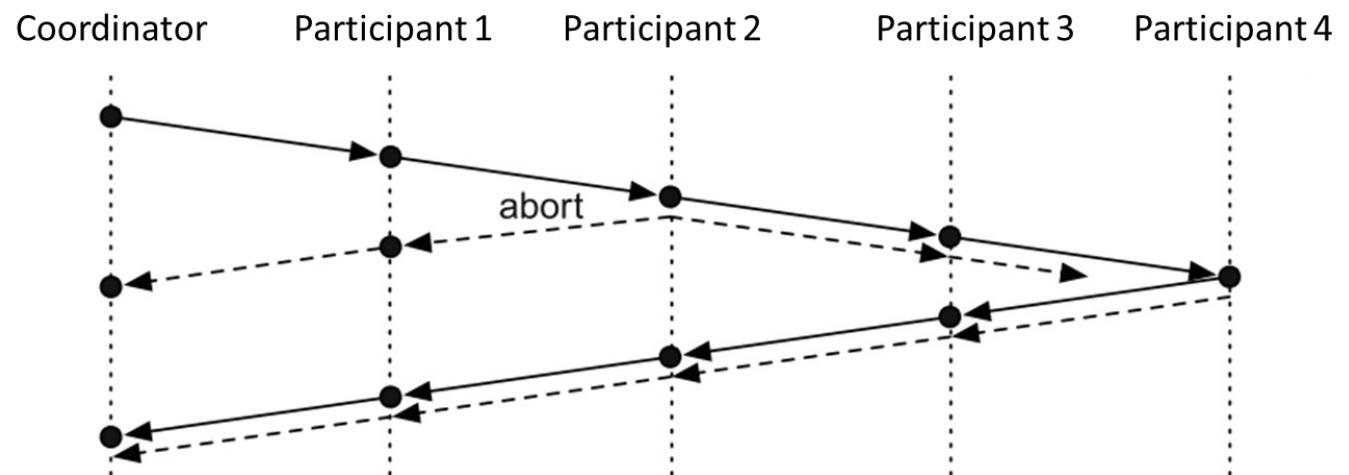
- Sequential messaging from participant to participant for voting
- Coordinator is just initiating the process

Cons

- Results in long response times
- No parallelism

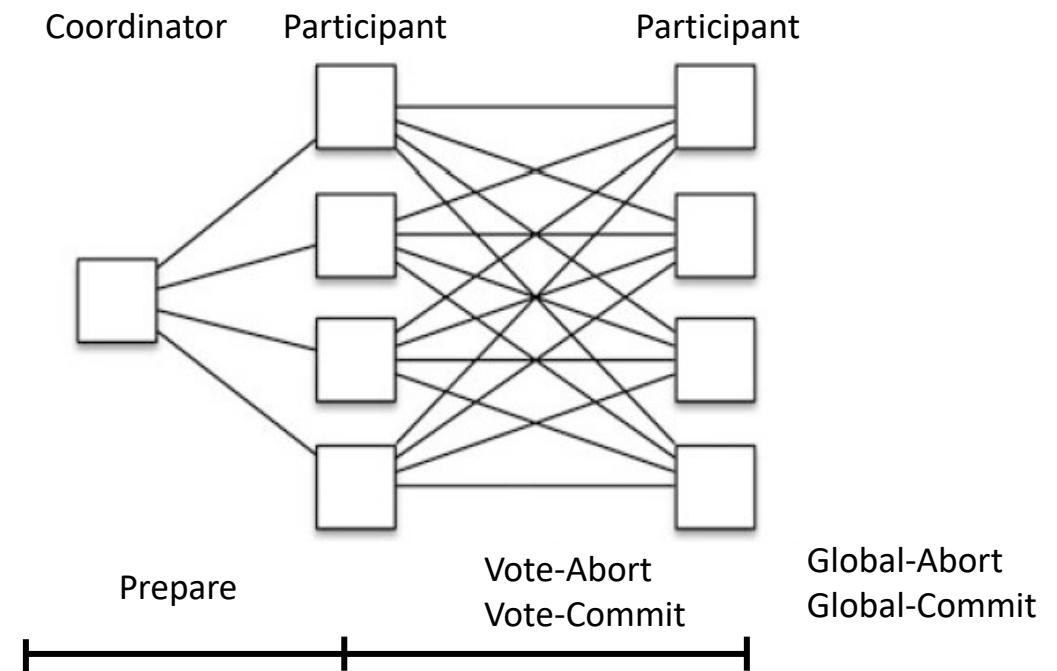
Pros

- Minimal number of messages



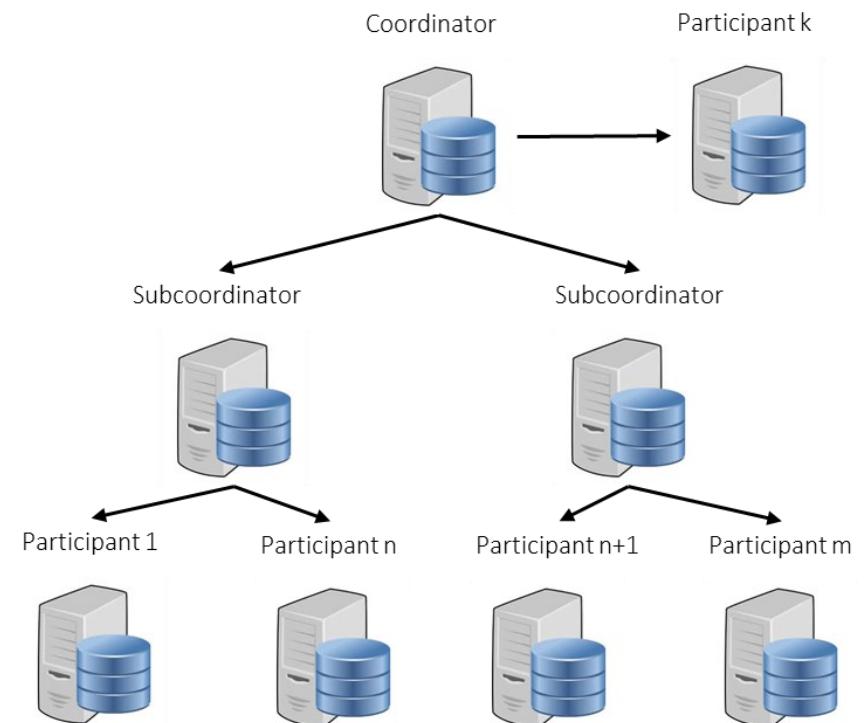
Distributed 2PC Protocol

- Voting is executed locally at the participants
- Coordinator sends prepare message to all participants
- Participants write decision to log, send vote-commit/vote-abort to all other participants
- If a participant receives all voting results → can finalize the operation (commit/abort)
- Pro: no coordinator needed to finish, short response times
- Con: communication overhead



Hierarchical 2PC Protocol

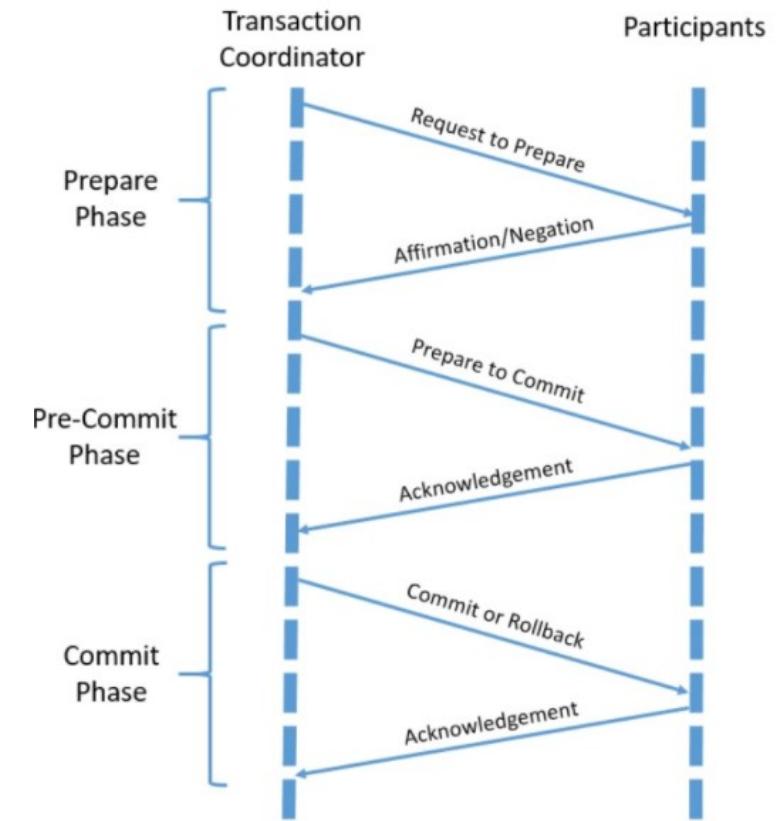
- The most common one in practice (e.g., DB2, MySQL)
- Suited for geographically highly distributed system to reduce communication costs
- Hierarchical execution with subcoordinators, which are coordinators of their subsystem
- Fits nicely to the structure of tx trees



3 Phase Commit Protocol

- Disadvantages 2PC:
 - blocking states may occur, e.g., when the coordinator is failing before all participants got a global decision → wait forever
- Hence, a 3rd phase Pre-Commit is introduced
 - First phase is equal to 2PC, also for abort
 - Specialty: if all vote-commit → log **pre-commit** and send to all participants → coordinator cannot abort tx anymore
 - Participants acknowledge with PC-ACK
 - Coordinator received k acks of $n-1$ participants → global commit
 - If coordinator fails → new coordinator is chosen

PAXOS Protocol as famous version → Google Spanner, Bing



Synchronisation of Distributed TXs

- **Goals**
 - *Transparent multi-user mode*
 - Guarantee consistency even if there are parallel transactions
- **Anomalies:** dirty read, lost update, non-repeatable read, phantom problem
- **Serializability:** scheduler executes TX in parallel as if they were executed in series
- **Global serializability:** system-wide, on all TX (global or local)
→ Need specific locking protocols!



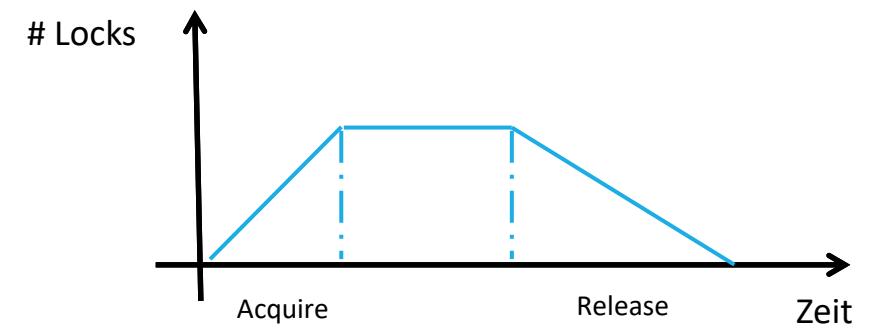
2 Phase Locking Protocol – 2 Variants

- **Centralized:** A central server manages the locks

- **Pros:** Server knows all locks, synchronization as in the usual RDBMS incl. deadlock detection
- **Cons:** Communication overhead, node autonomy violated, single point of failure

- **Distributed:** Each node manages its own locks

- **Pros:** No separate communication necessary, release of locks is managed by commit protocol
- **Cons:** Complicated handling of global deadlocks



Timestamp Protocol

- Check serializability locally with timestamps
- Each TX gets a local timestamp at BOT → need clock synchronization
- Write and read timestamps: **wts, rts** → timestamp of the last TX, which wrote/read the data object

Conflicts

- A TX t_2 is not allowed to access object x when in conflict with a more recent write operation of TX t_1
$$ts(t_2) < wts(x)$$
- Analogously, TX t_2 is not allowed to write an object x, if a more recent tx has read or written the object
$$ts(t_2) < \max(rts(x), wts(x))$$
- If one of the conditions is fulfilled (conflict) → t_2 is aborted and restarted



- Which TX must be aborted and restarted?

Time	T_1	T_2	T_3
1	$w(y)$		
2		$r(y)$	
3			$r(x)$
4	$r(x)$		
5		$w(y)$	
6	$r(y)$		

References & Further Reading

Parts of the slides are based on course material by

- Prof. Dr. Matthias Jarke (Information Systems and Databases, RWTH Aachen University)
- Prof. Dr. Christoph Quix (Wirtschaftsinformatik und Data Science, Hochschule Niederrhein)

Further Reading

[Elmasri & Navathe, 2017] Elmasri, R., & Navathe, S. (2017). *Fundamentals of database systems* (Vol. 7). Pearson

[Kemper & Eickler, 2015] Kemper, A., & Eickler, A. (2015). *Datenbanksysteme*. Oldenbourg Wissenschaftsverlag

[Rahm et al., 2015] Rahm, E., Saake, G., & Sattler, K. U. (2015). *Verteiltes und paralleles Datenmanagement*. Springer Berlin Heidelberg.