

Data bases 2

Instructors: Sara Comai, Piero Fraternali, Davide Martinenghi

sara.comai@polimi.it

piero.fraternali@polimi.it

davide.martinenghi@polimi.it

Data bases 2, 2023/2024

- Teachers:
 - Piero Fraternali
 - piero.fraternali@polimi.it
 - 02-2399-3640
 - Dipartimento di Elettronica Informazione e Bioingegneria, 1st floor
- Exercise sessions:
 - Nicolò Oreste Pincioli Vago
nicolooreste.pincioli@polimi.it

Data bases 2, 2023/2024

089183 - DATA BASES 2		5.00	1	Milano Leonardo		149							
Scaglioni da manifesto		Orario didattico											
Data	Dove	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00	17:00	18:00	19:00	20:00
Lunedì	2.1.2												
		DATA BASES 2 lezione (dal 18/09/2023 al 18/12/2023)											
Martedì													
Mercoledì													
Giovedì													
Venerdì	20.S.1												
		DATA BASES 2 esercitazione (dal 15/09/2023 al 22/12/2023)											

- Attendance is only in presence (no live streaming)
- All lessons are recorded (they were already recorded last year, index of links to videos is provided in WeBeep)

Textbooks

- IN ITALIAN:
 - P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone “Basi di dati: Architetture e linee di evoluzione” (2003)
 - P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone “Basi di dati” (2018)
 - EBOOK available at VitalSource Bookshelf (<https://resolver.vitalsource.com/>)
- IN ENGLISH:
 - P. Atzeni, S. Ceri, S. Paraboschi, R. Torlone “Database systems” McGraw-Hill (1999)
 - NOW DOWNLOADABLE (<http://dbbook.dia.uniroma3.it/>)
 - Compared to the last Italian version some chapters are missing
 - More bibliography on specific topics (e.g., JPA) listed in the official program page of the course

Teaching material

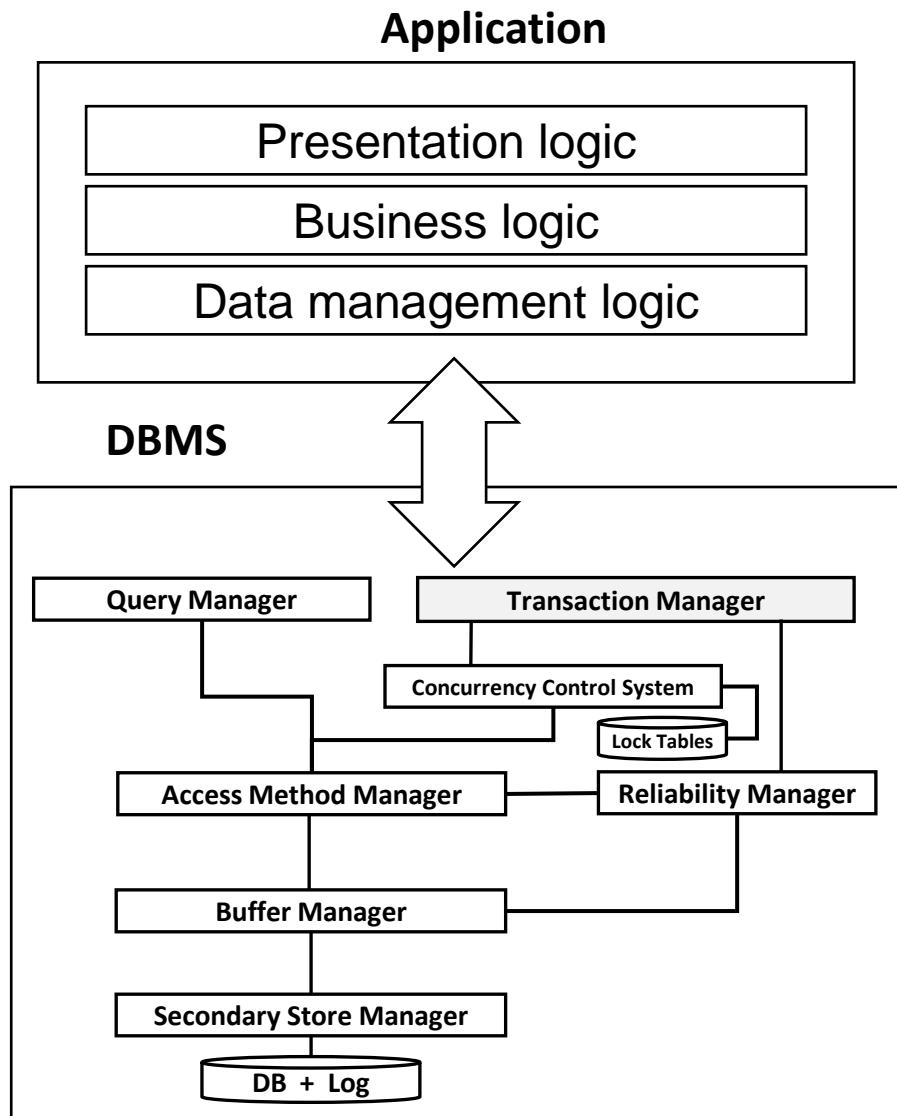
- Material is available on the Beep portal
<https://webeep.polimi.it/>
- under “Data bases 2”
- Materials include slides of the lectures, exercise sessions, forum, etc.

Prerequisites

- Basics of Database systems
 - Relational model
 - SQL (SQL-92)
 - Relational algebra
 - Object oriented design with UML and programming with Java (basic)
 - ONLINE TEXT BOOK COVERING DATABASE FUNDAMENTALS: <http://dbbook.dia.uniroma3.it/>

What you will learn from the course

- Opening the database box
 - Transaction management
 - Concurrency control
 - Reliability manager
 - Physical data management and query optimization
 - Data distribution
- Opening the application box
 - Mapping objects to tables and methods to transactions
 - Managing the alignment between object state in main memory and data state in secondary memory



DataBase Management System — DBMS

- A system (software product) capable of managing data collections that are:
 - **Large**: much larger than the central memory available on the computers that run the software
 - **Persistent**: with a lifetime which is independent of single executions of the programs that access them
 - **Shared**: used by several applications at a time
 - **Reliable**: ensuring tolerance to hardware and software failures
 - **Data ownership respectful**: by disciplining and controlling accesses

Technology of DBMSs – questions

- **Transaction management**
 - How do I ensure that a set of updates is executed in an “all or nothing” fashion? How do I undo the whole set if one update fails? → Acid properties
- **Concurrency control**
 - How do I ensure that my transaction executes “correctly” in the presence of other transactions that update the same data? → CC theory, pessimistic and optimistic locking
- **Reliability control**
 - How do I avoid data losses in the presence of failures? → Log and recovery protocols
- **Distributed architectures**
 - How can I detect deadlocks in distributed databases?
- **Buffer and secondary memory management**
 - How and when are data moved from main to secondary storage? → paging and caching
- **Physical data structures and access structures**
 - What are the low level data structures that store data? → sequential, hash-based and tree-based structures
- **Query management**
 - How can data organization be exploited to speed up queries? → cost-based optimization

Database-Application integration: questions

- How do I address the impedance mismatch between database and application models (Table vs Class, Tuple vs Object, Pointers vs Relations and Foreign Keys)?
→ code level procedure, Object Relational Mapping
- How do I make my application communicate with the database?
→ call level interface, ODBC-JDBC, JPA Persistence Provider
 - How can I align the state of an object and the state of the persistent data that correspond to it?
 - How do I map changes to objects to changes to persistent data?
→ JPA managed entities
- How can I ensure that application functions (e.g., object methods) execute with the proper “all or nothing” semantics in the database

Ranking data: questions

- We are able to store lots of data, but how do we get the *best* data out of a dataset?
- How do we characterize preferences over data?
 - What does “best” even mean in the case of tuples described by different attributes, as is the case with databases?
- How do we compute such data in distributed scenarios?
- Can we interpret data and preferences from a geometrical perspective?

→ All these questions are addressed by *multi-objective optimization*, i.e., the simultaneous optimization of several criteria (like attributes of a tuple) so as to *rank* data from best to worst
- How does SQL accommodate ranking?

DB Evolution

Since the 70's: relational databases + SQL

Some revolutions in the 90's:

- SQL'92
- SQL'99 (triggers, object-oriented features)

And more recently:

- 1996-2001: seminal research works on ranking in DBs and skylines
- SQL:2003 (XML-related features)
- SQL:2006 (XQuery)
- 2009: JPA final release
- SQL:2011 (Temporal DB)
- SQL:2016 (row pattern matching, JSON)
- Since 2005: NoSQL DBMS (no standard!)
- A single application may involve different kinds of data → diverse data models and query languages for a single application

Related topics addressed in other courses

- **056895 - STREAMING DATA ANALYTICS**
 - NOSQL systems
 - graph, column, document, key-value based storage; persistent and volatile solutions
 - BASE transaction models
 - Stream data storage: time series, data streams and events data bases and processing pipelines

Popularity of the models

420 systems in ranking, August 2023

Rank	DBMS			Database Model	Score		
	Aug 2023	Jul 2023	Aug 2022		Aug 2023	Jul 2023	Aug 2022
1.	1.	1.	Oracle	Relational, Multi-model	1242.10	-13.91	-18.70
2.	2.	2.	MySQL	Relational, Multi-model	1130.45	-19.89	-72.40
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	920.81	-0.78	-24.14
4.	4.	4.	PostgreSQL	Relational, Multi-model	620.38	+2.55	+2.38
5.	5.	5.	MongoDB	Document, Multi-model	434.49	-1.00	-43.17
6.	6.	6.	Redis	Key-value, Multi-model	162.97	-0.80	-13.43
7.	↑ 8.	↑ 8.	Elasticsearch	Search engine, Multi-model	139.92	+0.33	-15.16
8.	↓ 7.	↓ 7.	IBM Db2	Relational, Multi-model	139.24	-0.58	-17.99
9.	9.	9.	Microsoft Access	Relational	130.34	-0.38	-16.16
10.	10.	10.	SQLite	Relational	129.92	-0.27	-8.95
11.	11.	↑ 13.	Snowflake	Relational	120.62	+2.94	+17.50
12.	12.	↓ 11.	Cassandra	Wide column, Multi-model	107.38	+0.86	-10.76
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	98.65	+2.55	-15.24
14.	14.	14.	Splunk	Search engine	88.98	+1.87	-8.46
15.	↑ 16.	15.	Amazon DynamoDB	Multi-model	83.55	+4.75	-3.71
16.	↓ 15.	16.	Microsoft Azure SQL Database	Relational, Multi-model	79.51	+0.55	-6.67
17.	17.	17.	Hive	Relational	73.35	+0.48	-5.31
18.	18.	↑ 22.	Databricks	Multi-model	71.34	+2.87	+16.72

Exam

- The exam consists of a written verification covering all the topics of the course
 - 4 exercises on the 5 main topics, possibly with related theoretical questions
 - Main topics:
 1. DB architecture/technologies (concurrency control etc.)
 2. Triggers
 3. ORM and JPA
 4. Physical DBs/Query optimization
 5. Ranking and skylines

Exam rules

- During the exam:
 - No books, notes, or electronic devices are allowed
 - Cheating policies: Communication with other students is strictly prohibited. In this case, students will be asked to exit the classroom (both!).
 - In the event of plagiarism, students are notified to the appropriate offices and are postponed to a next exam *session*
- Positive marks:
 - If a mark is rejected and the written exam is retaken (simply sitting down and seeing the exam text is sufficient), the previous mark is lost!
- Negative marks
 - After the exam, we publish the exam solutions
 - If students are certain about failing the test, they are invited to withdraw
 - If we correct the exam and the grade is **below 10**, we record the grade **REJECTED (RP - RIPROVATO)**, i.e. the student is postponed to the next exam *session*
- Remote exams: if eligible, you have to find a supervisor of the foreign institution for the surveillance and notify the teacher at least one week before the exam date
- Please, check the general rules for the exams at Politecnico:
<https://www.polimi.it/en/current-students/didactic-activities-and-degree-examination/exams>

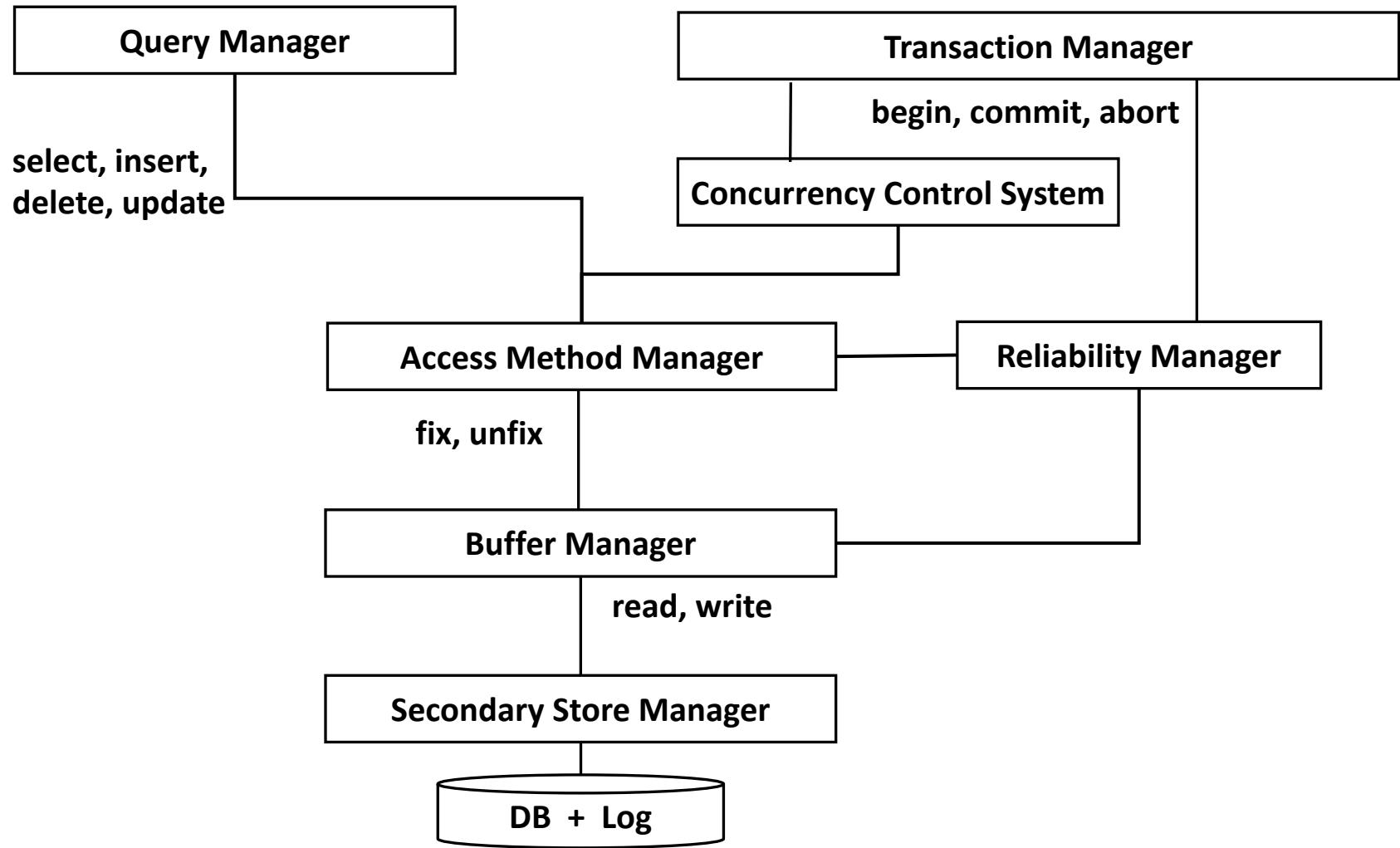
Data bases 2

Transactional Systems

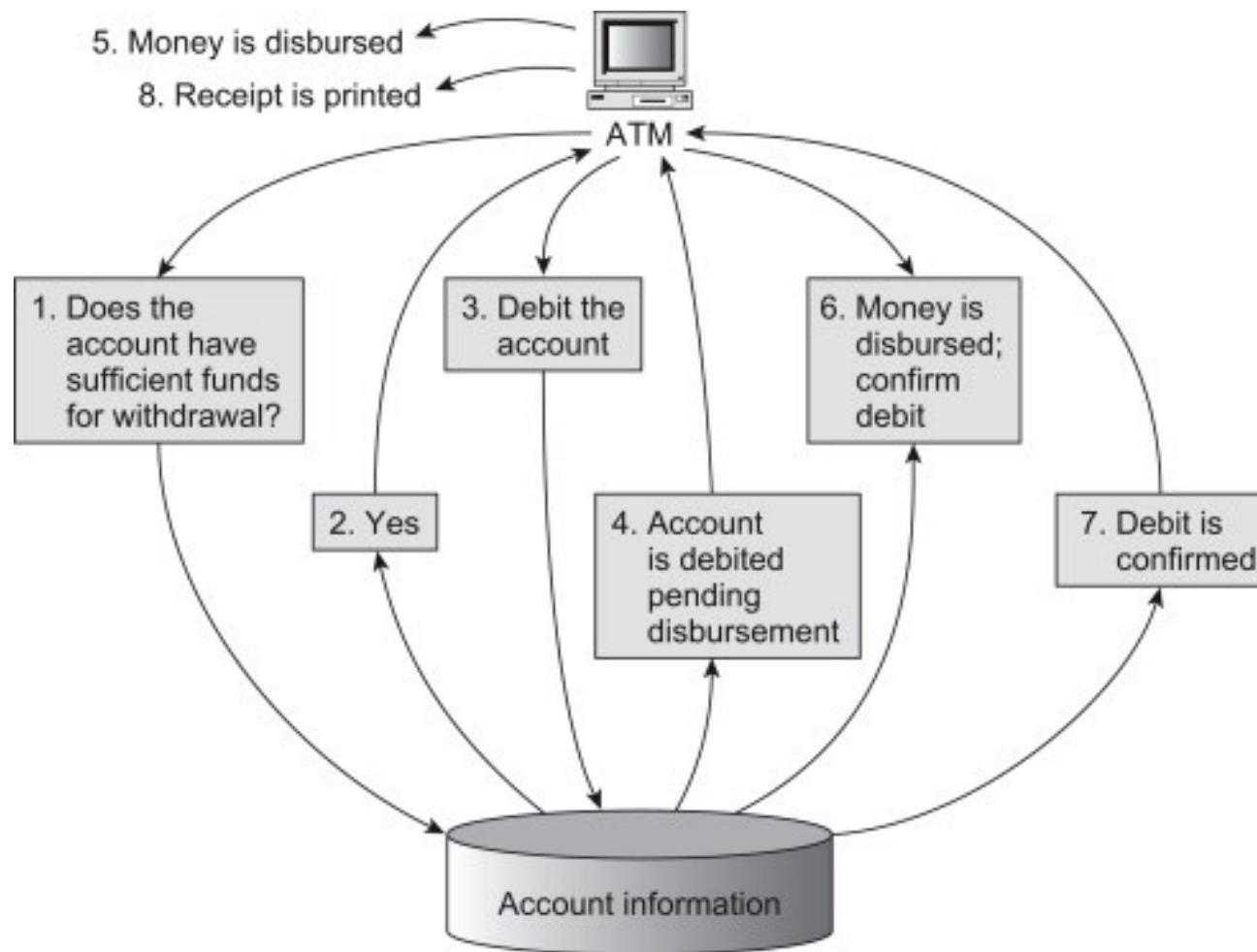
In this lecture

- The architecture of the DDBMS
- Introduction to the concept of transaction
- Properties of a transaction
- Modules of the DBMS responsible of guaranteeing such properties

The architecture of the DBMS



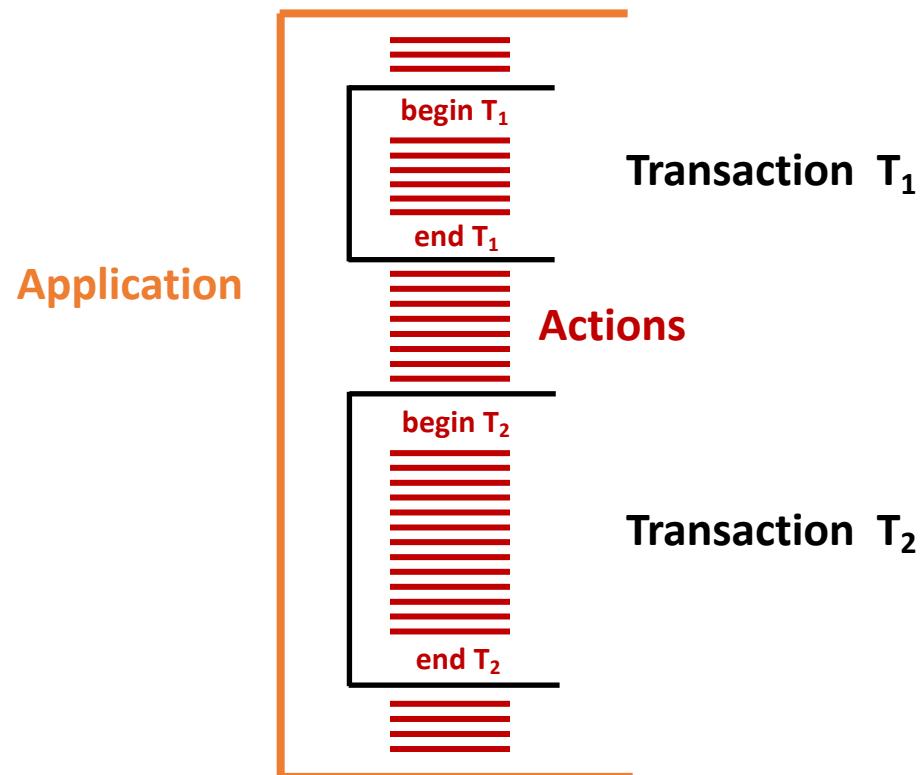
The need of transactions: ATM example



Definition of Transaction

- An elementary, atomic unit of work performed by an application
- Each transaction is conceptually encapsulated within two commands:
 - begin transaction
 - end transaction
- Within a transaction, one of the commands below is executed (exactly once) to signal the end of the transaction:
 - commit-work (commit)
 - rollback-work (abort)
- Transactional System (OLTP): a system that supports the execution of transactions on behalf of concurrent applications

Difference between Application and Transaction



Transactions: bank transfer

```
begin transaction;
```

```
update Account  
set Balance = Balance + 10  
where AccNum = 12202;
```

```
update Account  
set Balance = Balance - 10  
where AccNum = 42177;
```

```
commit-work; // end transaction
```

Transactions: bank transfer upon condition

```
begin transaction;  
update Account  
set Balance = Balance + 10 where AccNum = 12202;  
  
update Account  
set Balance = Balance - 10 where AccNum = 42177;  
  
select Balance into A from Account  
where AccNum = 42177;  
  
if ( A >= 0 ) then  
commit-work; // end transaction with success  
else  
rollback-work; // end transaction with failure
```

ACID Properties of Transactions

- A transaction is a unit of work enjoying the following properties:
 - Atomicity
 - Consistency
 - Isolation
 - Durability

Atomicity

- A transaction is an indivisible unit of execution
 - Either all the operations in the transaction are executed or none is executed
- In the case of the bank transfer, the execution of a single update statement would be disastrous
- The time in which **COMMIT** is executed marks the instant in which the transaction ends successfully:
 - An error before should cause the rollback of the work
 - An error afterwards should not alter the effect of the transaction
- The **ROLLBACK** of the work performed can be caused
 - By the application with a ROLLBACK statement
 - By the DBMS, for example for the violation of integrity constraints or for concurrency management
- In case of a rollback, the work performed must be undone, bringing the database to the state it had before the start of the transaction
- It is the application's responsibility to decide whether an aborted transaction must be redone or not

Consistency

- A transaction must satisfy the DB integrity constraints
 - if the initial state S_0 is consistent
 - then the final state S_f is also consistent
 - This is not necessarily true for the intermediate states S_i
- Example
 - The sum of the worked hours per task should equal the planned work hours of the project
 - If the constraint holds before the transaction it must hold also after its execution
 - The constraint can be temporarily violated during the execution of the transaction, e.g., when shifting work from a task to another one, but must be satisfied at the end

Isolation

- The execution of a transaction must be independent of the concurrent execution of other transactions
- In particular, the concurrent execution of a number of transactions **must produce the same result as the execution of the same transactions in a sequence**
- E.g., the concurrent execution of T1 and T2 must produce the same results that can be obtained by executing one of these sequences
 - T1,T2
 - T2,T1
- Isolation impacts performance and trade-offs can be defined between isolation and performance

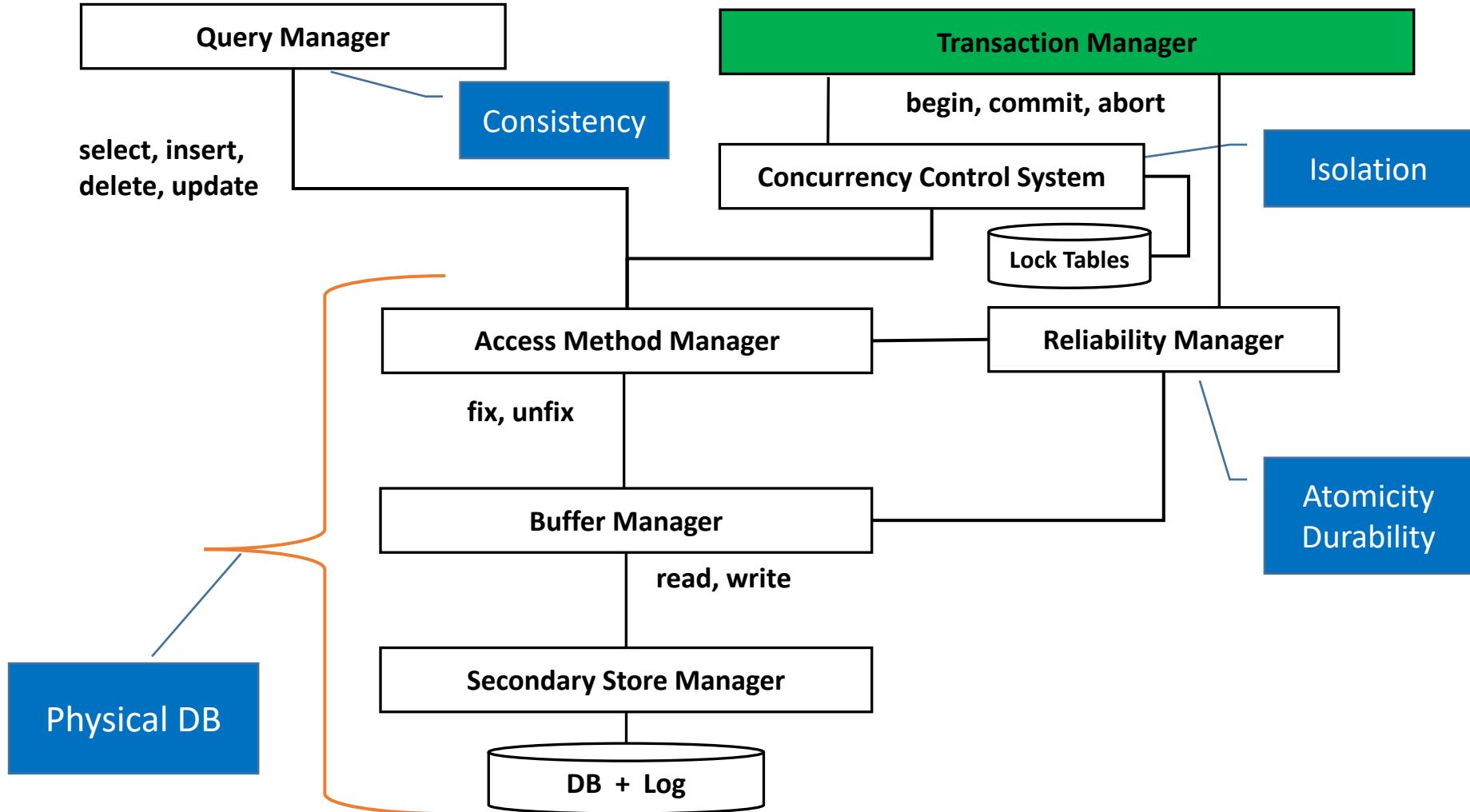
Durability

- The effect of a transaction that has successfully committed will last "forever"
 - Independently of any system fault
 - Sounds obvious... but every DBMS manipulates data in main memory

ACID properties & DBMS mechanisms

- **Atomicity**
 - Abort-rollback-restart, commit protocols
 - Reliability Manager
- **Consistency**
 - Integrity checking of the DBMS
 - Integrity Control System at query execution time (with the support of the DDL compiler)
- **Isolation**
 - Concurrency control
 - Concurrency Control System
- **Durability**
 - Recovery management
 - Reliability Manager

Transactions & DBMS architecture



Transactions in Java + JDBC

```
public void addExpenseReport(ExpenseReport expenseReport, Mission mission)
                            throws SQLException, BadMissionForExpReport {
    //Check that the mission exists and is in OPEN state
    if (mission == null | mission.getStatus() != MissionStatus.OPEN) {
        throw new BadMissionForExpReport("Mission cannot introduce expense report");
    }
    MissionsDAO missionDAO = new MissionsDAO(connection);
    String query = "INSERT into expenses (food, accom, transp, mission) VALUES(?, ?, ?, ?)";
    // Delimit the transaction explicitly
    connection.setAutoCommit(false);
    try (PreparedStatement pstatement = connection.prepareStatement(query)) {
        pstatement.setDouble(1, expenseReport.getFood());
        pstatement.setDouble(2, expenseReport.getAccomodation());
        pstatement.setDouble(3, expenseReport.getTransportation());
        pstatement.setInt(4, expenseReport.getMissionId());
        pstatement.executeUpdate(); // 1st update
        // 2nd update to change the status of the mission, in a separate component
        missionDAO.changeMissionStatus(expenseReport.getMissioId(), MissionStatus.REPORTED);
        connection.commit();
    } catch (SQLException e) {
        connection.rollback(); // if update 1 OR 2 fails, roll back all work
        throw e;
    }
} finally { connection.setAutoCommit(true); }
```

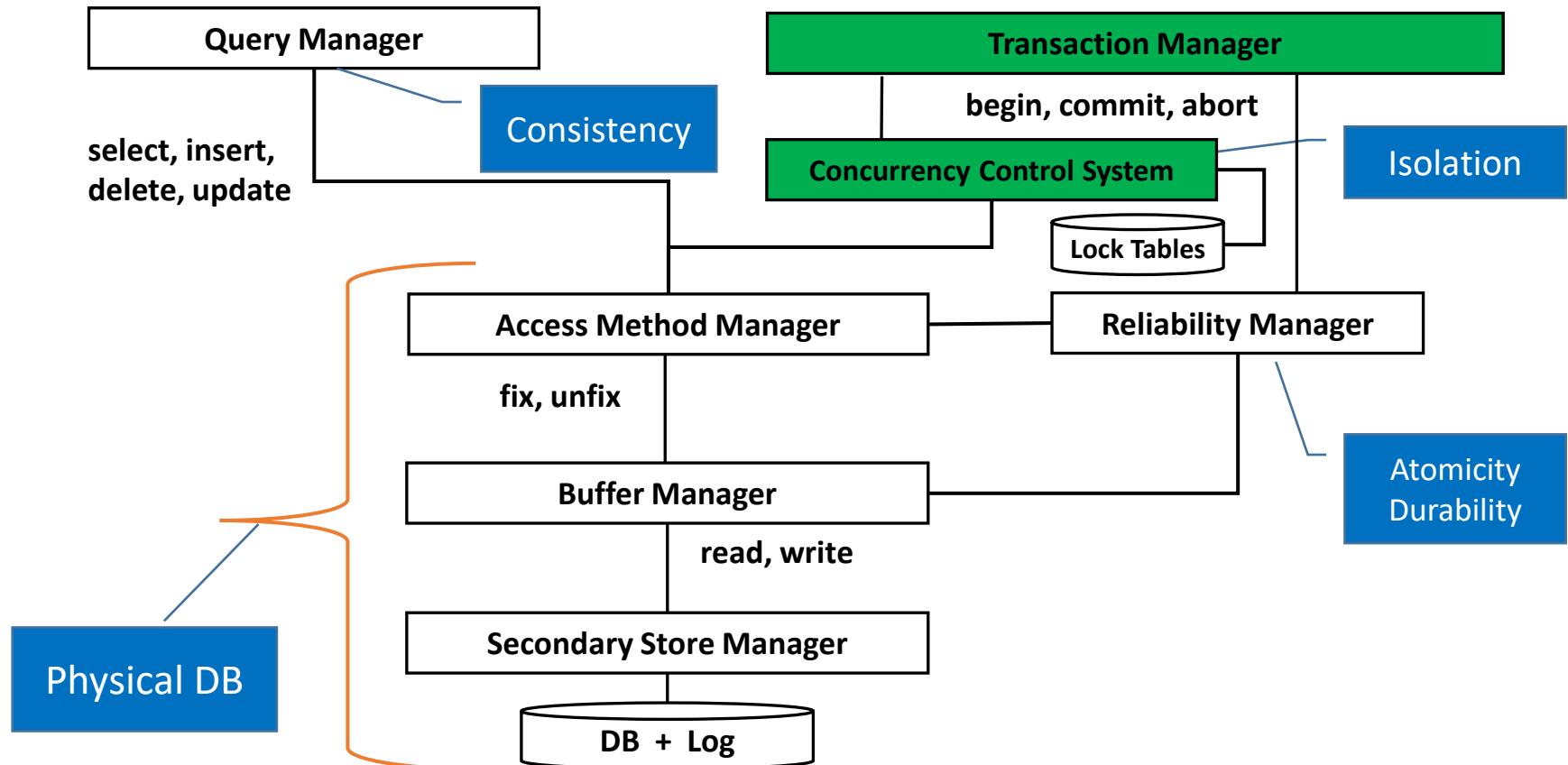
Transaction management in JDBC: issues

- This style of programming transactions is known as “programmatic demarcation”
- The programmer writes explicit code to start / end the transaction
- Issues:
 - What happens if I forget to handle an exception?
 - What happens if a component A starts a transaction and then calls a component B that also starts a transaction?
 - Error? Nested Transactions? Joined transaction?
 - How can an application method know if a transaction is active when it is executed?
- Some middleware systems (e.g., in Java EE) help automate the management of transactions by the application

Data bases 2

Concurrency Control

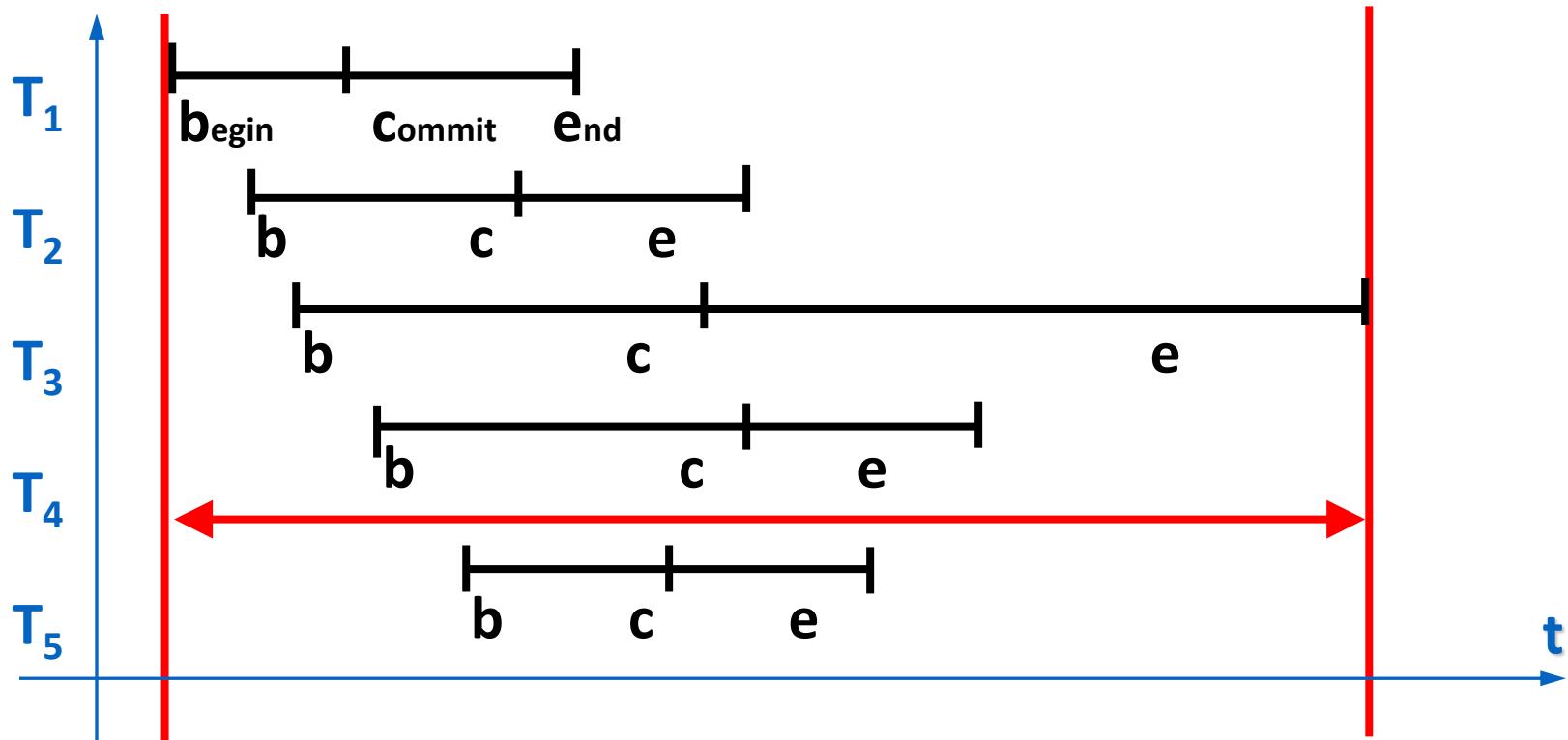
Concurrency in the DBMS architecture



- **The Concurrency Control System**

- Manages the simultaneous execution of transactions
- Avoids the insurgence of anomalies
- While ensuring performance

Advantages of Concurrency



- Goal: exploit parallelism to maximise transactions per second (TPS)

Concurrency Control

- Concurrency is fundamental
 - Tens or hundreds of transactions per second cannot be executed serially
- Examples: banks, ticket reservations
- Problem: concurrent execution may cause anomalies
 - Concurrency needs to be controlled

Problems due to Concurrency

T₁: **begin transaction**

```
update account  
set balance = balance + 3  
where customer = 'Smith'  
commit work
```

T₂: **begin transaction**

```
update account  
set balance = balance + 6  
where customer = 'Smith'  
commit work
```

Concurrent SQL transactional statements addressing the same resource

T₁: **begin transaction**

```
D = D + 3  
commit work
```

T₂: **begin transaction**

```
D = D + 6  
commit work
```

T₁: **begin transaction**

```
var x;  
read(D → x)  
x = x + 3  
write(x → D)  
commit work
```

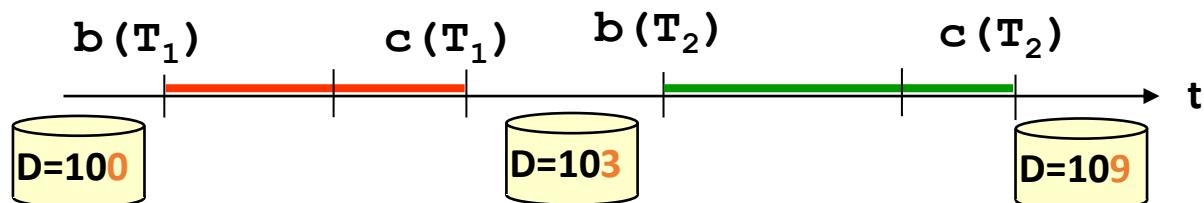
T₂: **begin transaction**

```
var y;  
read(D → y)  
y = y + 6  
write(y → D)  
commit work
```

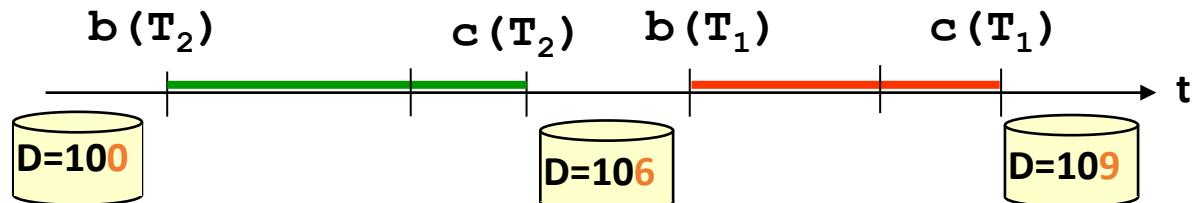
Serial Executions

```
bot(T1)
var x;
r(D → x)
x = x + 3
w(x → D)
commit(T1)
```

Initially D₀=100

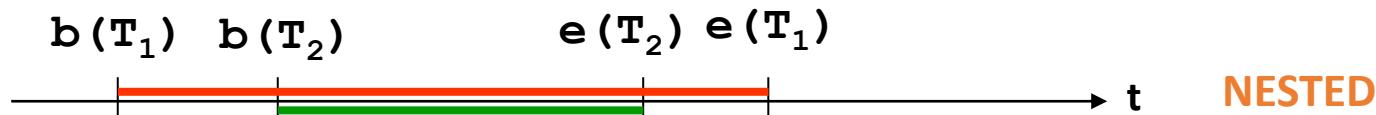
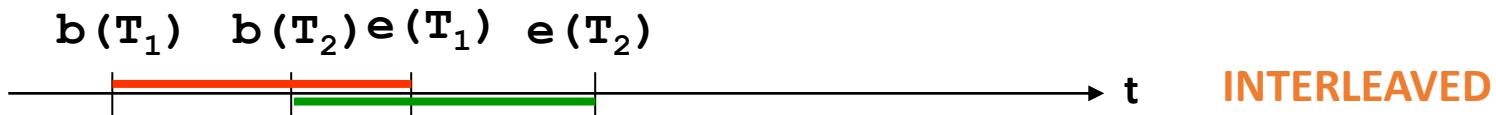
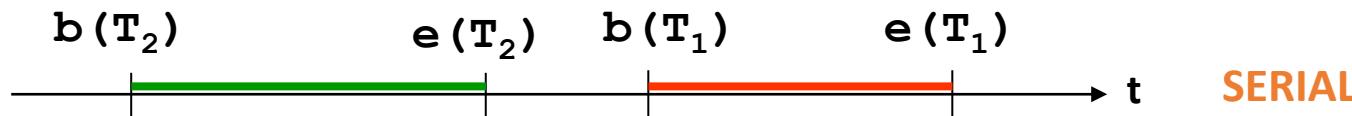
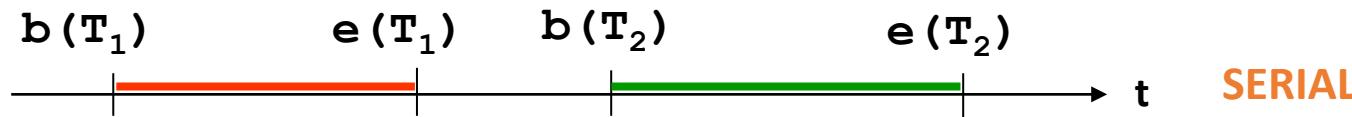


```
bot(T2)
var y;
r(D → y)
y = y + 6
w(y → D)
commit(T2)
```



Note: it is **not** important that the final value be the same

Concurrent Executions



Execution with Lost Update

$D = 100$

1 $T_1: r(D \rightarrow x)$

2 $T_1: x = x + 3$

3 $T_2: r(D \rightarrow y)$

4 $T_2: y = y + 6$

5 $T_1: w(x \rightarrow D)$

6 $T_2: w(y \rightarrow D)$

x and y have the
same (initial)
value

$D = 103$

$D = 106 !$

T1 : UPDATE account

SET balance = balance + 3

WHERE client = 'Smith'

T2 : UPDATE account

SET balance = balance + 6

WHERE client = 'Smith'

- Note: this anomaly does not depend merely on T2 overwriting the value produced by T1
 - $w_1(x), w_2(x)$ is ok (serial)
 - $r_1(x), w_1(x), w_2(x)$ is ok too (serial)
 - $r_1(x), r_2(x), w_1(x), w_2(x)$ is not ok: inconsistent updates **from the same initial value**

Sequence of I/O actions producing the error



or



Dirty Read

D = 100

1 T₁: r(D → x)

2 T₁: x = x + 3

3 T₁: w(x → D) D = 103

4 T₂: r(D → y) this read is “dirty” (uncommitted value)

5 T₁: **rollback**

6 T₂: y = y + 6

7 T₂: w(y → D) D = 109 !

T1 : UPDATE account

SET balance = balance + 3

WHERE client = ‘Smith’

T2 : UPDATE account

SET balance = balance + 6

WHERE client = ‘Smith’

Nonrepeatable Read

$D = 100$

1 T_1 : $r(D \rightarrow x)$

2 T_2 : $r(D \rightarrow y)$

3 T_2 : $y = y + 6$

4 T_2 : $w(y \rightarrow D)$ $D = 106$

5 T_1 : $r(D \rightarrow z)$ $z \neq x !$

T1 : **SELECT balance FROM account
WHERE client = 'Smith'**

...

**SELECT balance FROM account
WHERE client = 'Smith'**

T2 : **UPDATE account
SET balance = balance + 6
WHERE client = 'Smith'**

Phantom Update

Constraint: $A+B+C=100$, $A=50$, $B=30$, $C=20$

$T_1: r(A \rightarrow x), r(B \rightarrow y) \dots$

$T_2: r(B \rightarrow s), r(C \rightarrow t)$

$T_2: s = s + 10, t = t - 10$

$T_2: w(s \rightarrow B), w(t \rightarrow C)$ (now $B=40$, $C=10$, $A+B+C=100$)

$T_1: r(C \rightarrow z)$ (but, for T_1 , $x+y+z = A+B+C = 90!$)

- So for T_1 it is as if “somebody else” had updated the value of the sum
- But for T_2 the update is perfectly legal (does not change the value of the sum)

Phantom Insert

Tab	
A	B

T₁: C=AVG(B: A=1) C = select avg(B) from Tab where A=1

T₂: Insert (A=1,B=2) insert into Tab values(1, 2)

T₁: C=AVG(B: A=1) C = ...

- Note: this anomaly does not depend on data already present in the DB when T1 executes, as in non repeatable read, but on a “phantom” tuple that is inserted by “someone else” and satisfies the conditions of a previous query of T1

Summary of Anomalies

- Lost update $r_1 - r_2 - w_2 - w_1$
 - An update is applied from a state that ignores a preceding update, which is lost
- Dirty read $r_1 - w_1 - r_2 - \text{abort}_1 - w_2$
 - An uncommitted value is used to update the data
- Nonrepeatable read $r_1 - r_2 - w_2 - r_1$
 - Someone else updates a previously read value
- Phantom update $r_1(\text{partial}) - r_2(\text{partial}) - w_2 \text{ (partial)} - r_1$
 - Someone else updates data that contributes to a previously valid constraint
- Phantom insert $r_1 - w_2(\text{new data}) - r_1$
 - Someone else inserts data that contributes to a previously read datum

Concurrency Theory vs System Implementation

- **Model:** an abstraction of a system, object or process, which purposely disregards details to simplify the investigation of relevant properties
- Concurrency Theory builds upon a model of transactions and concurrency control principles that helps understanding real systems
- Real systems exploit implementation level mechanisms (locks, snapshots) which help achieve some of the desirable properties postulated by the theory
 - Do not look for a view or conflict serializability checker in your DBMS
 - Look instead for lock tables, lock types, lock granting rules, snapshots, etc..
 - Understand how the implementation mechanisms ensure properties modelled by the Concurrency Theory

Transactions, Operations, Schedules

- Operation: a read or write of a specific datum by a specific transaction
 - NB: the schedule notation omits the program variables
 - $r_1(x) = r_1(x \rightarrow .)$
 - $w_1(x) = w_1(. \rightarrow x)$
 - $r_1(x)$ and $r_1(y)$ are different operations
 - $r_1(x)$ and $r_2(x)$ are different operations
 - $w_1(x)$ and $w_2(x)$ are different operations
- Schedule: a sequence of operations performed by concurrent transactions that respects the order of operations of each transaction
 - T1: $r_1(x) w_1(x)$
 - T2: $r_2(z) w_2(z)$
 - S1: $r_1(x) r_2(z) w_1(x) w_2(z)$

Schedules

- How many distinct schedules exist for two transactions?

- With T1: $r1(x)$ $w1(x)$ T2: $r2(z)$ $w2(z)$

$$N = 6$$

- $r1(x) \quad w1(x) \quad r2(z) \quad w2(z)$
- $r2(z) \quad w2(z) \quad r1(x) \quad w1(x)$
- $r1(x) \quad r2(z) \quad w1(x) \quad w2(z)$
- $r2(z) \quad r1(x) \quad w2(z) \quad w1(x)$
- $r1(x) \quad r2(z) \quad w2(z) \quad w1(x)$
- $r2(z) \quad r1(x) \quad w1(x) \quad w2(z)$

} *serial*

} *interleaved*

} *nested*

Schedules: how many?

- How many **serial** schedules exist for n transactions? N_S
- How many **distinct** schedules exist for n transactions? N_D
- n transactions ($T_1, T_2, \dots, T_i, \dots, T_n$), each with k_i operations
- T_1 has k_1 operations, T_i has k_i operations...

$T_1: o_{11} o_{12} \dots o_{1k_1}$

$$N_S = n!$$

$T_2: o_{21} o_{22} \dots o_{2k_2}$

...

$T_i: o_{i1} o_{i2} \dots o_{ik_i}$

...

$T_n: o_{n1} o_{n2} \dots o_{nk_n}$

$$N_S << N_D$$

- N_S = number of permutations of N transactions
- N_D = number of permutations of all operations /
product of the number of permutations of the operations of
each transaction (only 1 out of the k_i permutations is valid,
the one that respects the sequence of operations)

Principles of Concurrency Control

- Goal: to reject schedules that cause anomalies
- **Scheduler:** a component that accepts or rejects the operations requested by the transactions
- **Serial schedule:** a schedule in which the actions of each transaction occur in a contiguous sequence
- S: r0(x) r0(y) w0(x) r1(y) r1(x) w1(y) r2(x) r2(y) r2(z) w2(z)

Principles of Concurrency Control

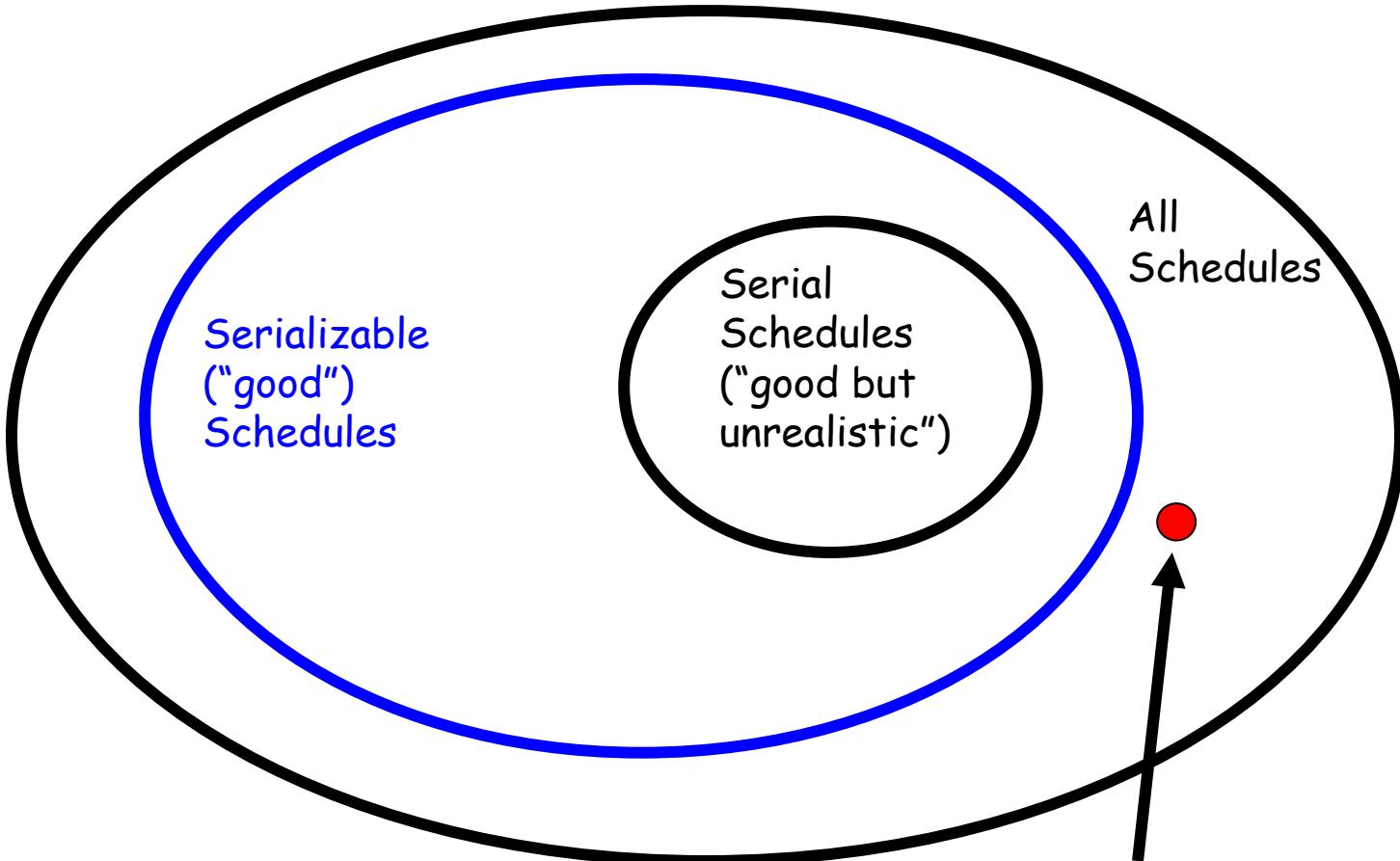
- **Serializable schedule**

- A schedule that leaves the database in the same state as **some** serial schedule of the same transactions
- Commonly accepted as a notion of schedule **correctness**
- Requires a notion of schedule equivalence to identify classes of schedules that ensure serializability
 - Different notions → different classes (cost of checking if schedule is serializable)

- Assumption

- We initially assume that transactions are observed “*a-posteriori*” and limited to those that have committed (**commit-projection**), and we decide whether the observed schedule is admissible
- In practice (and in contrast), schedulers must make decisions **while transactions are running**

Basic Idea



This schedule **may** generate anomalies.
Its execution leads to a database state that
no serial execution would produce

View-serializability

- Preliminary definitions:
 - $r_i(x)$ **reads-from** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ in S between $r_i(x)$ and $w_j(x)$
 - $w_i(x)$ in a schedule S is a **final write** if it is the last write on x that occurs in S
- Two schedules are **view-equivalent** ($S_i \approx_v S_j$) if: they have 1) the same operations, 2) the same reads-from relationship, and 3) the same final writes
- A schedule is **view-serializable** if it is view-equivalent to a serial schedule of the same transactions
- The class of view-serializable schedules is named **VSR**
- **Mnemonically:** S is view-serializable if 1) every read operation sees the same values and 2) the final value of each object is written by the same transaction as if the transactions were executed serially in **some** order

Note

- The reads-from relationship $r_i(x)$ reads-from $w_j(x)$ in a schedule S assumes that $r_i(x)$ reads the value written by $w_j(x)$ independently of the time at which the commit of T_j occurs
 - In other words the value written by $w_j(x)$ could be uncommitted when $r_i(x)$ reads it but we are sure (by definition of commit projection) that it will be committed

Examples of view-serializability

- **S3:** w0(x) **r2(x)** r1(x) **w2(x)** w2(z)
- S4: w0(x) **r1(x)** **r2(x)** **w2(x)** w2(z) --- **SERIAL**
- (switch of read operations on the same object)
- S3 is view-equivalent to serial schedule S4 (so it is view-serializable)

- **S5:** w0(x) r1(x) w1(x) **r2(x)** w1(z)
- S6: w0(x) r1(x) **w1(x)** w1(z) **r2(x)** --- **SERIAL**
- (switch of read/write ops on different objects that preserves final write)
- S5 is not view-equivalent to S4 (different operations) but is view-equivalent to serial schedule S6, so it is also view-serializable

Examples of view-serializability

- S7 : r1(x) r2(x) w1(x) w2(x) // lost update pattern
- S8 : r1(x) r2(x) w2(x) r1(x) // non-repeatable read pattern
- S9 : r1(x) r1(y) r2(z) r2(y) w2(y) w2(z) r1(z) // phantom update pattern
- They are all **non** view-serializable (check by testing view equivalence with T1, T2 and T2, T1)
- Example with S7 = r1(x) **r2(x)** w1(x) **w2(x)**:
 - **T1,T2** = r1(x) w1(x) **r2(x)** w2(x): in S7 T2 reads the initial value of x, not the one written by T1
 - **T2,T1** = r2(x) w2(x) r1(x) **w1(x)**: in S7 T2 makes the final write on x, not T1

A More Complex Example

$S_{10} : w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ \textcolor{red}{r_2(x)} \ w_0(y) \ r_3(z) \ w_3(z) \ w_2(y) \ \textcolor{blue}{w_1(x)}$
 $w_3(y)$

- Is S_{10} serializable?
 - Yes iff there exists a serial schedule S_s s.t. $S_{10} \approx_v S_s$
- Let's try with $S_{11} : T_0 \ T_1 \ T_2 \ T_3$
 - $w_0(x) \ w_0(z) \ w_0(y) \ r_1(x) \ r_1(z) \ \textcolor{blue}{w_1(x)} \ \textcolor{red}{r_2(x)} \ w_2(y) \ r_3(z) \ w_3(z) \ w_3(y)$
- is **not** view equivalent to S_{10}

A More Complex Example

S_{10} : $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

- Let's try with S_{12} : T_0 T_2 T_1 T_3
- S_{12} : $w_0(x)$ $w_0(z)$ $w_0(y)$ $r_2(x)$ $w_2(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$ $w_3(y)$

reads-from OK: $r_1(x)$ from $w_0(x)$,
 $r_1(z)$ from $w_0(z)$,
 $r_2(x)$ from $w_0(x)$,
 $r_3(z)$ from $w_0(z)$,

final writes OK: $w_1(x)$, $w_3(y)$, $w_3(z)$

$$S_{10} \in VSR$$

A More Complex Example

S_{10} : $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $w_0(y)$ $r_3(z)$ $w_3(z)$ $w_2(y)$ $w_1(x)$ $w_3(y)$

- Let's try with S_{13} : T0,T2,T3,T1
- S_{13} : $w_0(x)$ $w_0(z)$ $w_0(y)$ $r_2(x)$ $w_2(y)$ $r_3(z)$ $w_3(z)$ $w_3(y)$ $r_1(x)$ $r_1(z)$ $w_1(x)$

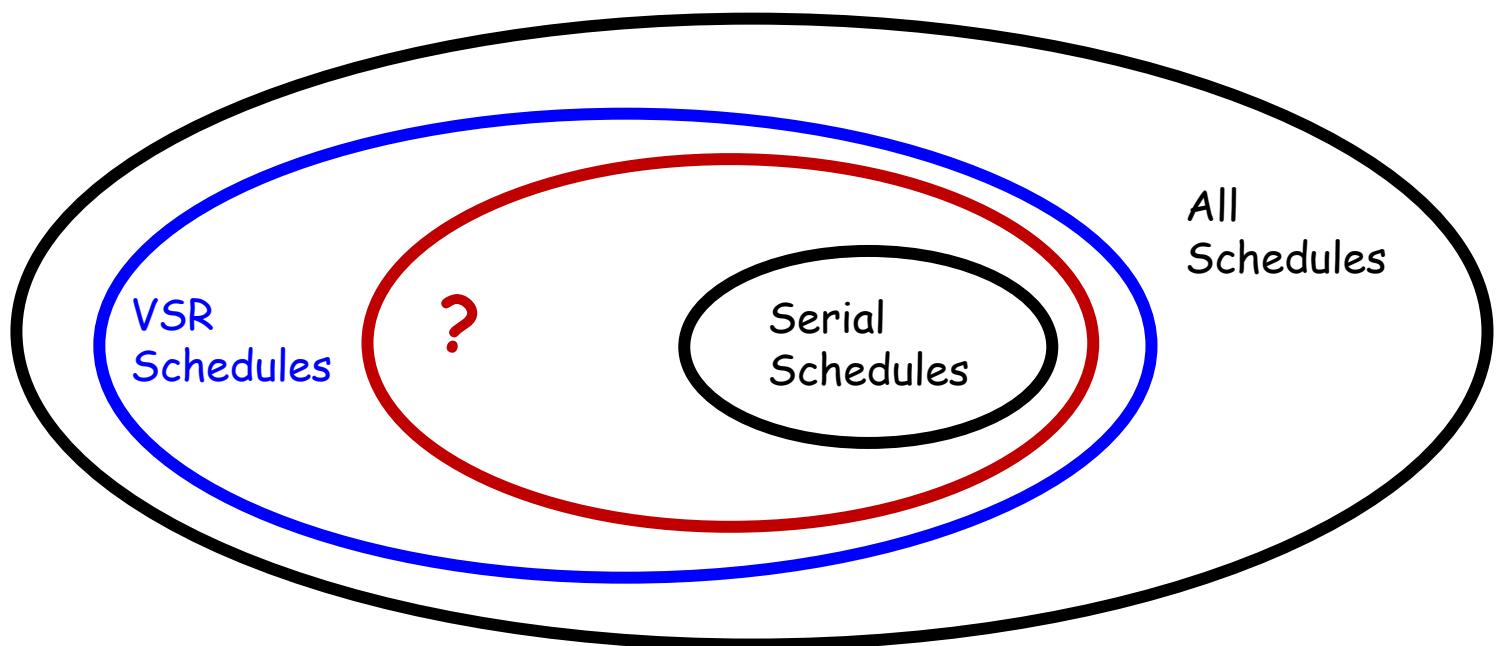
reads-from : $r_1(x)$ from $w_0(x)$, S_{10} not view-equivalent to T0,T2,T3,T1
 $r_1(z)$ from $w_3(z)$

Different reads-from relationship

Complexity of view-serializability

- Deciding view-equivalence of two given schedules is done in polynomial time and space
- Deciding if a **generic** schedule is in VSR is an **NP-complete problem**
 - requires considering the reads-from and final writes of all possible serial schedules with the same operations – combinatorial in the general case
 - Performance!! what can we trade for that?
 - ...Accuracy!
- We look for a **stricter definition that is easier to check**
 - May lead to **rejecting some schedule that would be acceptable under view-serializability** but not under the stricter-faster criterion

VSR schedules are "too many"



Conflict-serializability

- Preliminary definition:
 - Two operations o_i and o_j ($i \neq j$) are in **conflict** if they address the same resource and at least one of them is a write
 - **read-write** conflicts (r-w or w-r)
 - **write-write** conflicts (w-w)

Conflict-serializability

- Two schedules are **conflict-equivalent** ($S_i \approx_C S_j$) if :
 - S_i and S_j contain the same operations and in all the conflicting pairs the transactions occur in the same order
- A schedule is **conflict-serializable** iff it is conflict-equivalent to a serial schedule of the same transactions
- The class of conflict-serializable schedules is named **CSR**

Relationship between CSR and VSR

- **VSR ⊃ CSR** : all conflict-serializable schedules are also view-serializable, but the converse is not necessarily true
- Proof: there are VSR schedules not in CSR
- Counter-example: we consider $r_1(x) w_2(x) w_1(x) w_3(x)$ that
 - is view-serializable: it is view-equivalent to

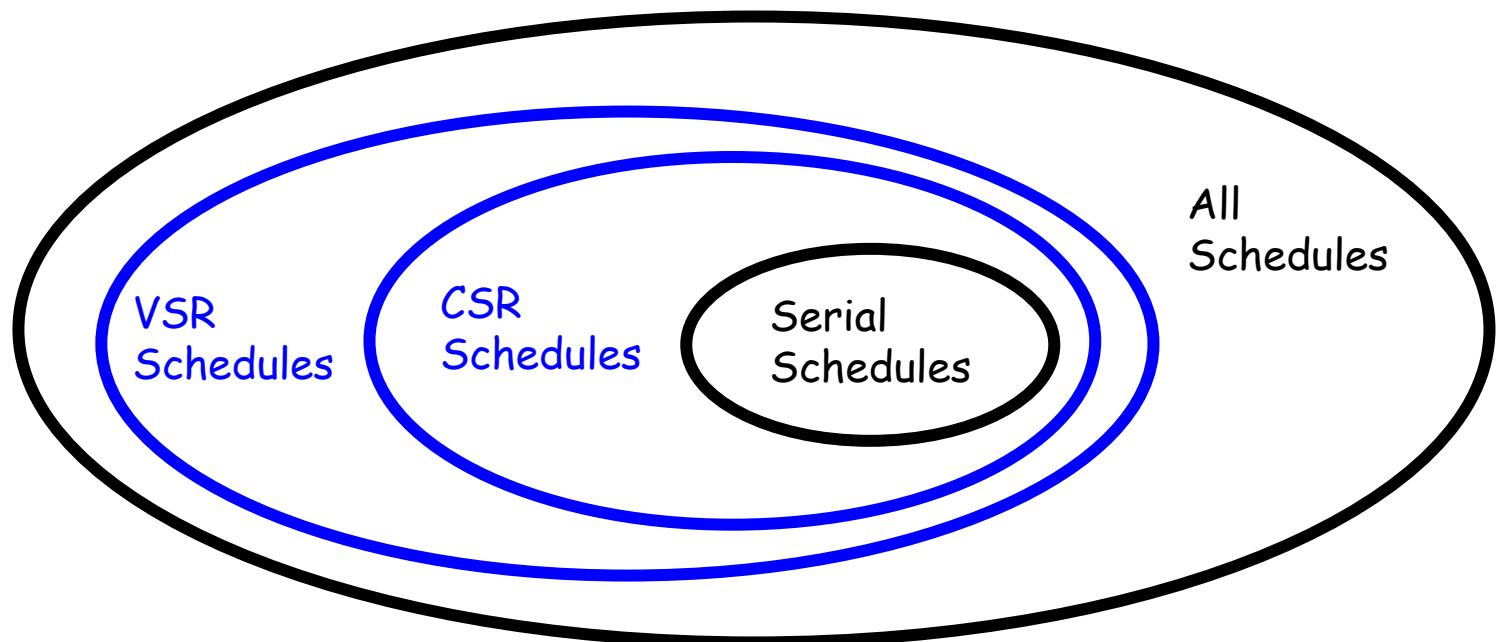
$$T_1 T_2 T_3 = r_1(x) w_1(x) w_2(x) w_3(x)$$

- is not conflict-serializable, due to the presence of
 $r_1(x) w_2(x)$ and $w_2(x) w_1(x)$
- there is no conflict-equivalent serial schedule
 - neither $T_1 T_2 T_3$ w.r.t. which it is view equivalent
 - nor $T_2 T_1 T_3$ which preserves the final write on x
 - nor all the others..

CSR implies VSR

- CSR → VSR: conflict-equivalence \approx_c implies view-equivalence \approx_v
- We assume $S1 \approx_c S2$ and prove that $S1 \approx_v S2$
- $S1$ and $S2$ must have:
 - The same final writes: if they didn't, there would be at least two writes in a different order, and since two writes are conflicting operations, the schedules would not be \approx_c
 - The same "reads-from" relations: if not, there would be at least one pair of conflicting operations in a different order, and therefore, again, \approx_c would be violated

CSR and VSR



Testing conflict-serializability

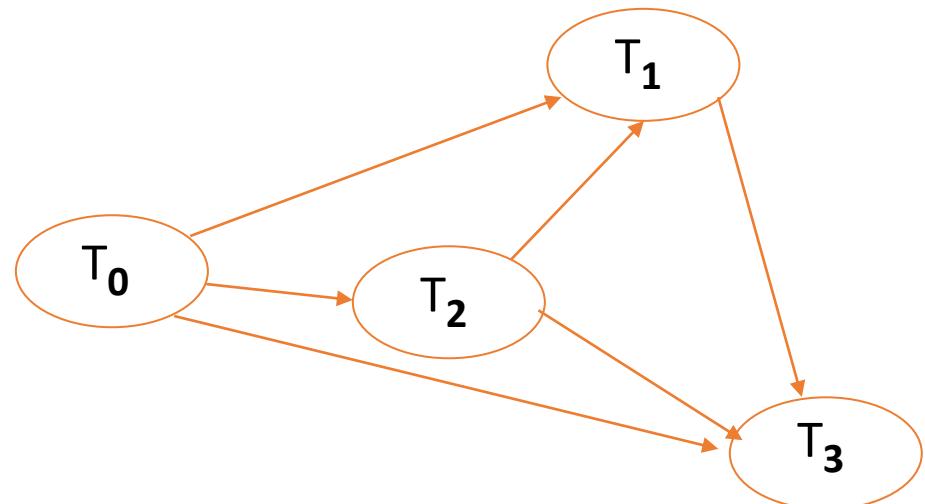
- Is done with a *conflict graph* that has:
 - One node for each transaction T_i
 - One arc from T_i to T_j if there exists at least one conflict between an operation o_i of T_i and an operation o_j of T_j such that o_i precedes o_j
- Theorem:
 - A schedule is in CSR if and only if its conflict graph is acyclic

Testing conflict-serializability

$S_{10} : w_0(x) \ r_1(x) \ w_0(z) \ r_1(z) \ r_2(x) \ w_0(y) \ r_3(z) \ w_3(z) \ w_2(y) \ w_1(x) \ w_3(y)$

- Resource-based projections:

- **x** : $w_0 \ r_1 \ r_2 \ w_1$
- **y** : $w_0 \ w_2 \ w_3$
- **z** : $w_0 \ r_1 \ r_3 \ w_3$

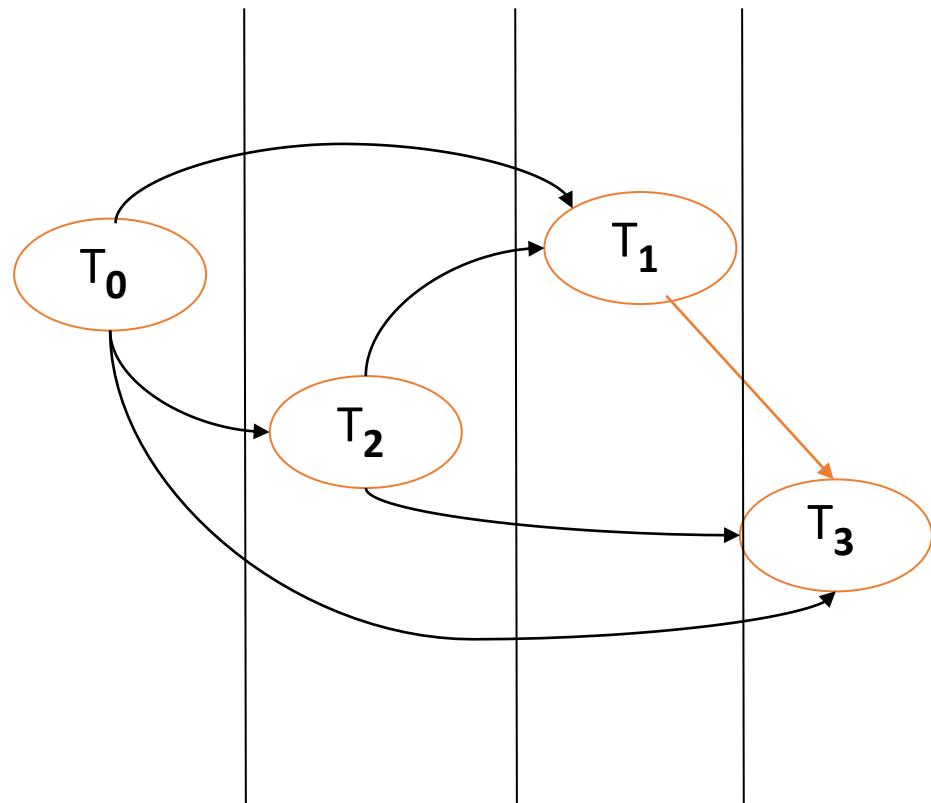


CSR implies acyclicity of the CG

- Consider a schedule S in CSR. As such, it is \approx_C to a serial schedule
- W.l.o.g. we can (re)label the transactions of S to say that their order in the serial schedule is: $T_1 T_2 \dots T_n$
- Since the serial schedule has all conflicting pairs in the same order as schedule S , in the conflict graph there can only be arcs (i,j) , with $i < j$
- Then the graph is acyclic, as a cycle requires at least an arc (i,j) with $i > j$

Acyclicity of the CG implies CSR

- If S 's graph is acyclic then it induces a topological (partial) ordering on its nodes, i.e., an ordering such that the graph only contains arcs (i,j) with $i < j$. The same partial order exists on the transactions of S
- Any serial schedule whose transactions are ordered according to the partial order is conflict-equivalent to S , because for all conflicting pairs (i,j) it is always $i < j$
 - In the example before: $T_0 < T_2 < T_1 < T_3$
 - In general, there can be many compatible serial schedules (i.e., many serializations for the same acyclic graph)
 - As many as the total orders compatible with the partial topological order



Let's go back...

- $r1(x) w2(x) w1(x) w3(x)$
 - it is VSR – it is view-equivalent to
- $T1 T2 T3 = r1(x) w1(x) w2(x) w3(x)$
 - it is not CSR, due to the presence of the cycle between $r1(x) w2(x)$ and $w2(x) w1(x)$
 - there is no conflict-equivalent serial schedule, neither $T1 T2 T3$ nor $T2 T1 T3$
 - Try to understand the relationship between CSR and VSR...

Concurrency Control in Practice

- CSR checking would be efficient if we knew the graph from the beginning — but we don't
- A scheduler must rather work “**online**”, i.e., decide for each requested operation whether to execute it immediately or to reject/delay it
- It is not feasible to maintain the conflict graph, update it, and check its acyclicity at each operation request
- The assumption that concurrency control can work only with the commit-projection of the schedule is unrealistic, **aborts do occur**
- Some simple on-line “decision criterion” is required for the scheduler, which must
 - **avoid as many anomalies as possible**
 - **have negligible overhead**

Arrival sequences vs a posteriori schedules

- So far the notation
 - $r_1(x) w_2(x) w_1(x) w_3(x)$
- represented a “schedule”, which is an **a posteriori view** of the execution of concurrent transactions in the DBMS (also called “history” in some books)
- A schedule represents “what has happened”, “which operations have been executed by which transaction in which order”
- They can be further restricted by the **commit-projection hypothesis** to operations executed by **committed** transactions
- When dealing with “online” concurrency control, it is important also to consider **“arrival sequences”**, i.e., sequences of operation requests emitted in order by transactions
- With an abuse of notation, we will denote an arrival sequence in the same way as a posteriori schedule
 - $r_1(x) w_2(x) w_1(x) w_3(x)$
- The distinction will be clear from the context
- The CC system maps an arrival sequence into an effective a posteriori schedule

Concurrency control approaches

- How can concurrency control be implemented “online”?
- Two main families of techniques:
 - Pessimistic
 - Based on locks, i.e., resource access control
 - If a resource is taken, make the requester wait or pre-empt the holder
 - Optimistic
 - Based on timestamps and versions
 - Serve as many requests as possible, possibly using out-of-date versions of the data
- We will compare the two families after introducing their features
- Commercial systems take the best of both worlds

Locking

- It's the most common method in commercial systems
- A transaction is **well-formed** w.r.t. locking if
 - read operations are preceded by **r_lock**
(aka **SHARED LOCK**) and followed by **unlock**
 - write operations are preceded by **w_lock**
(aka **EXCLUSIVE LOCK**) and followed by **unlock**
 - Note: unlocking can be **delayed** w.r.t. to the end of the read/write operation
- Transactions that first read and then write an object may:
 - Acquire a **w_lock** already when reading
 - Acquire a **r_lock** first and then upgrade it into a **w_lock** (lock escalation)
- Possible states of an object:
 - **free**
 - **r-locked** (locked by one or more readers)
 - **w-locked** (locked by a writer)

Behavior of the Lock Manager (Conflict Table)

- The lock manager receives the primitives from the transactions and grants resources according to the conflict table
 - When a lock request is granted, the resource is acquired
 - When an unlock is executed, the resource becomes available or the lock counter is decreased

REQUEST	RESOURCE STATUS		
	FREE	R_LOCKED	W_LOCKED
r_lock	OK R_LOCKED	OK R_LOCKED (n++)	NO W_LOCKED
w_lock	OK W_LOCKED	NO R_LOCKED	NO W_LOCKED
unlock	ERROR	OK DEPENDS (n--)	OK FREE

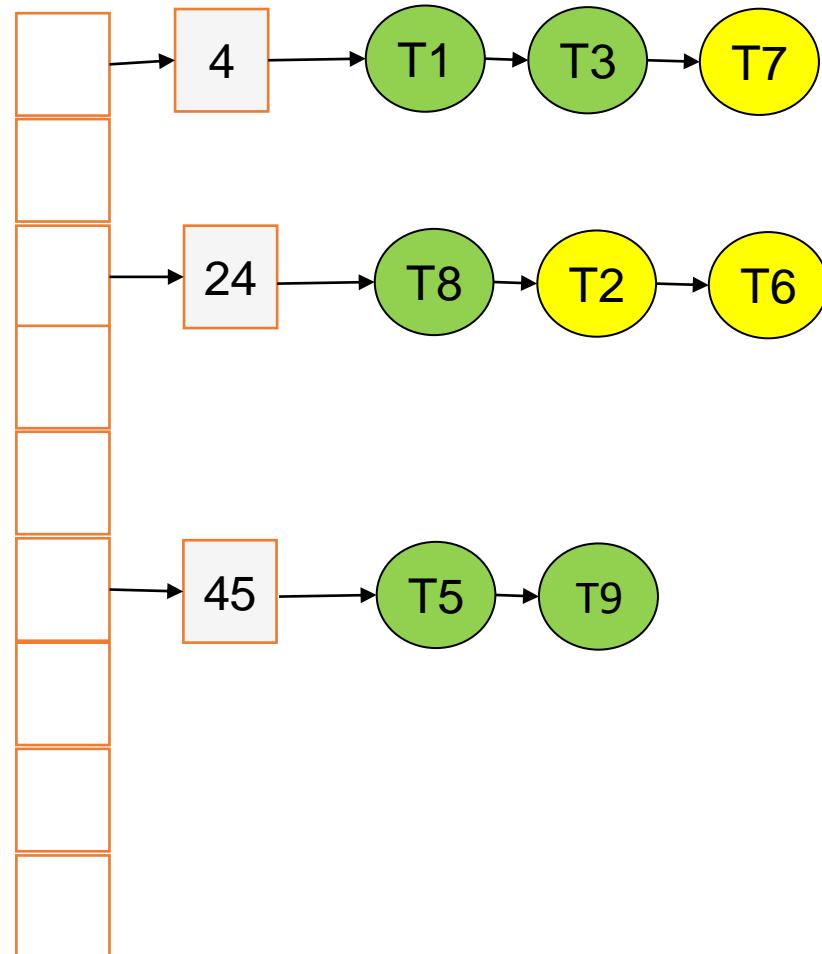
n: counter of the concurrent readers, (inc|dec)remented at each (r_|un)lock

Example

- **Arrival sequence:** $r1(x)$, $w1(x)$, $r2(x)$, $r3(y)$, $w1(y)$
- $r1(x)$
 - $r1\text{-lock}(x)$ request \rightarrow OK \rightarrow x state = r-locked, $n_x=1$
- $w1(x)$
 - $w1\text{-lock}(x)$ request \rightarrow OK (upgrade) \rightarrow x state = w-locked
- $r2(x)$
 - $r2\text{-lock}(x)$ request \rightarrow NO because w-locked \rightarrow **T2 waits**
- $r3(y)$
 - $r3\text{-lock}(y)$ request \rightarrow OK \rightarrow y state = r-locked, $n_y=1$ T3 unlock(y) \rightarrow , $n_y=0$ y state = free
- $w1(y)$
 - $w1\text{-lock}(y)$ request \rightarrow OK \rightarrow y state = w-locked, T1 unlock(x) \rightarrow $n_x=0$ x status = free, T1 unlock(y) \rightarrow y status = free
 - $r2(x)$ was waiting for $r2\text{-lock}(x)$ request \rightarrow OK
x state = r-locked, $n_x=1$...
- **A posteriori schedule:** $r1(x)$, $w1(x)$, $r3(y)$, $w1(y)$, $r2(x)$
- Note: the schedule depends on when T1 actually unlocks x. Several a posteriori schedules are possible
- T2 is **delayed**

How are locks implemented

- Typically by lock tables, which are hash tables indexing the lockable items via hashing
- Each locked item has a linked list associated with it
- Every node in the linked list represents the transaction which requested for lock, lock mode (SL/XL) and current status (granted/waiting).
- Every new lock request for the data item is appended as a new node to the list.
- Locks can be applied on both data and index records

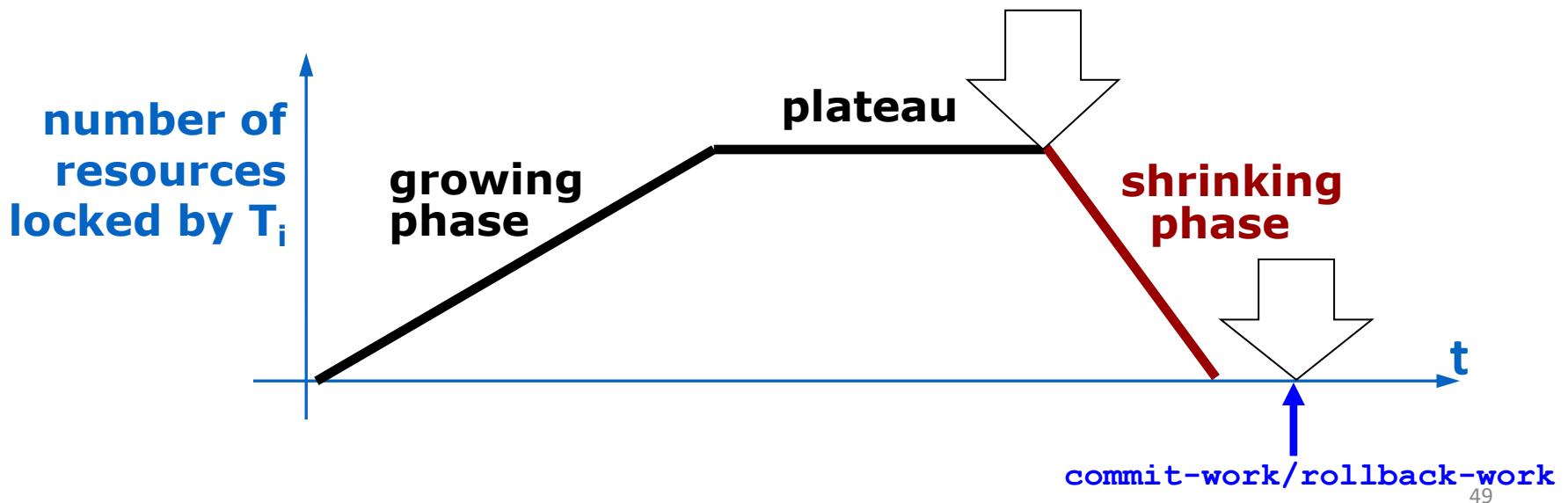


Is respecting locks enough for serializability?

- Arrival sequence $r_1(x), r_2(x), w_2(x), r_1(x)$
 - $r_1(x)$
 - $r_1\text{-lock}(x)$ request \rightarrow OK \rightarrow x state = r-locked, $n=1$ **T1 unlock(x)** \rightarrow x state = free, $n=0$
 - $r_2(x)$
 - $r_2\text{-lock}(x)$ request \rightarrow OK \rightarrow x state = r-locked, $n=1$
 - $w_2(x)$
 - $w_2\text{-lock}(x)$ request \rightarrow OK (upgrade) \rightarrow x state = w-locked
 - T2 unlock(x) \rightarrow x state = free, $n=0$
 - $r_1(x)$
 - **r1-lock(x)** request \rightarrow OK \rightarrow x state = r-locked , $n=1$..
- But the schedule mapped by the lock manager does not eliminate T1's **non repeatable read**
- T1 behavior: 1) releases its read lock on x too quickly; 2) acquires another lock after releasing one lock

Two-Phase Locking (2PL)

- Requirement (two-phase rule):
 - A transaction cannot acquire any other lock after releasing a lock
 - Sufficient to prevent non repeatable reads
 - But ensure stronger properties... serializability!



Serializability

- Consider a scheduler that
 - Only processes well-formed transactions
 - Grants locks according to the conflict table
 - Checks that all transactions apply the two-phase rule
- The class of generated schedules is called **2PL**
- Result: schedules in 2PL are both view- and conflict-serializable
- **(VSR ⊃ CSR ⊃ 2PL)**

2PL implies CSR

- CSR \supseteq 2PL: Every 2PL schedule is also conflict-serializable
- 2PL \rightarrow CSR:
 - Suppose a 2PL schedule S is not CSR
 - S's graph must contain at least one cycle $T_i \rightarrow T_j \rightarrow T_i$
 - Therefore there must be a pair of conflicting operations in reverse order, suppose that the conflictual operations appear as follows in the schedule
 - $OP_{hi}(x), OP_{kj}(x) \dots OP_{uj}(y), OP_{wi}(y)$ where one of $OP_{hi}(x), OP_{kj}(x)$ is a write and one of $OP_{uj}(y), OP_{wi}(y)$ is a write
 - $OP_{hi}(x), OP_{kj}(x)$
 - T_i must have released a lock for T_j to access x
 - Later in the schedule... $OP_{uj}(y), OP_{wi}(y)$
 - T_i must have acquired a lock for the conflict to occur \rightarrow CONTRADICTION
 - Inclusion of 2PL in VSR descends from $VSR \supseteq CSR$
 - This proves that all 2PL schedules are view-serializable too

2PL implies CSR (alternative proof)

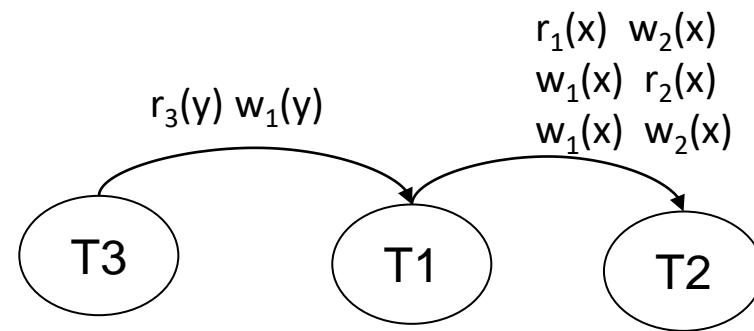
- 2PL → CSR: We assume that a schedule S is 2PL
- Consider, for each transaction, the end of the plateau
 - (i.e., the moment in which it holds all locks and is going to release the first one)
- We sort (and re-label) the transactions by this temporal value and consider the corresponding serial schedule S'
- In order to prove (by contradiction) that S is conflict-equivalent to S' , $S' \approx_C S$, ... (in all conflicting pairs transactions occur in the same order)

2PL implies CSR (alternative proof)

- Consider a (generic) conflict $o_i \rightarrow o_j$ in S' with $o_i \in T_i$ $o_j \in T_j$ $i < j$
- By definition of conflict, o_i and o_j address the same resource r , and at least one of them is a write
- Can o_i and o_j occur in reverse order in S ?
 - No, because then T_j should have released r before T_i could acquire it
 - This contradicts the ordering criterion (T_j should have started releasing locks before T_i)
- Inclusion of 2PL in VSR descends from $VSR \supseteq CSR$
- **This proves that all 2PL schedules are view-serializable too**
- **And they can be checked with negligible overhead**

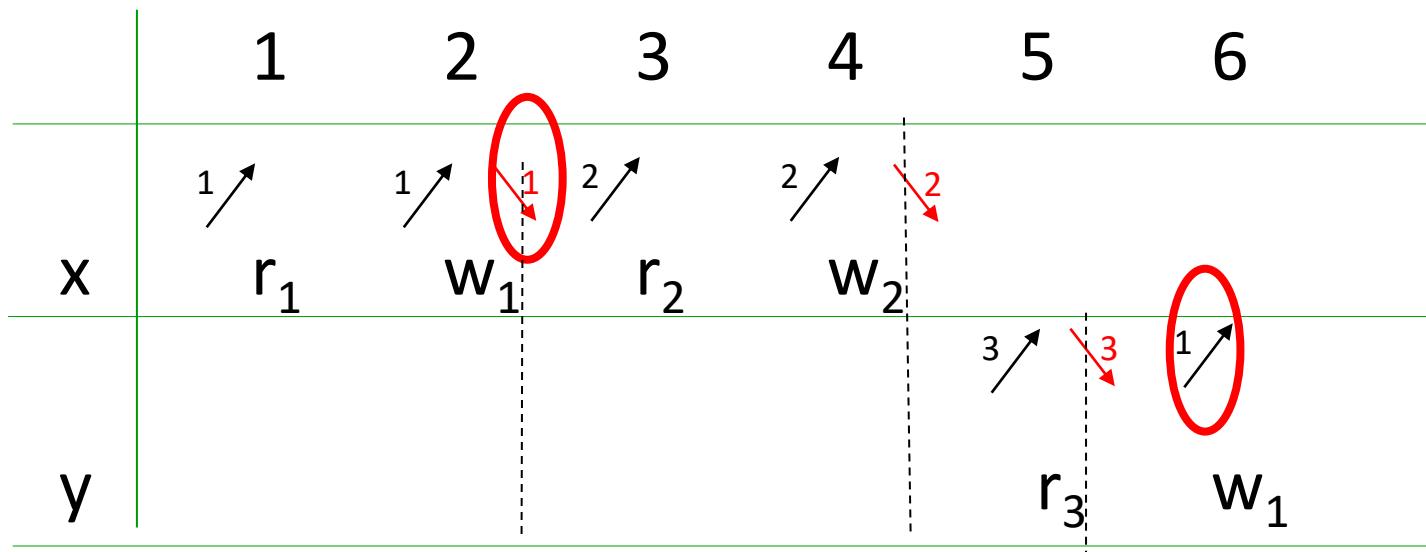
2PL smaller than CSR

- CSR \supset 2PL: Every 2PL schedule is also conflict-serializable, but the converse is not necessarily true
- Counter-example: $r_1(x) w_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$
 - It violates 2PL
 - $r_1(x) \text{ w}_1(x) \searrow r_2(x) \text{ w}_2(x) \text{ r}_3(y) \nearrow \text{w}_1(y)$
 - T1 releases T1 acquires
 - However, it is conflict-Serializable: $T_3 < T_1 < T_2$
 - On x: $r_1(x) w_1(x) r_2(x) w_2(x)$
 - On y: $r_3(y) w_1(y)$



A visualization of the 2PL test

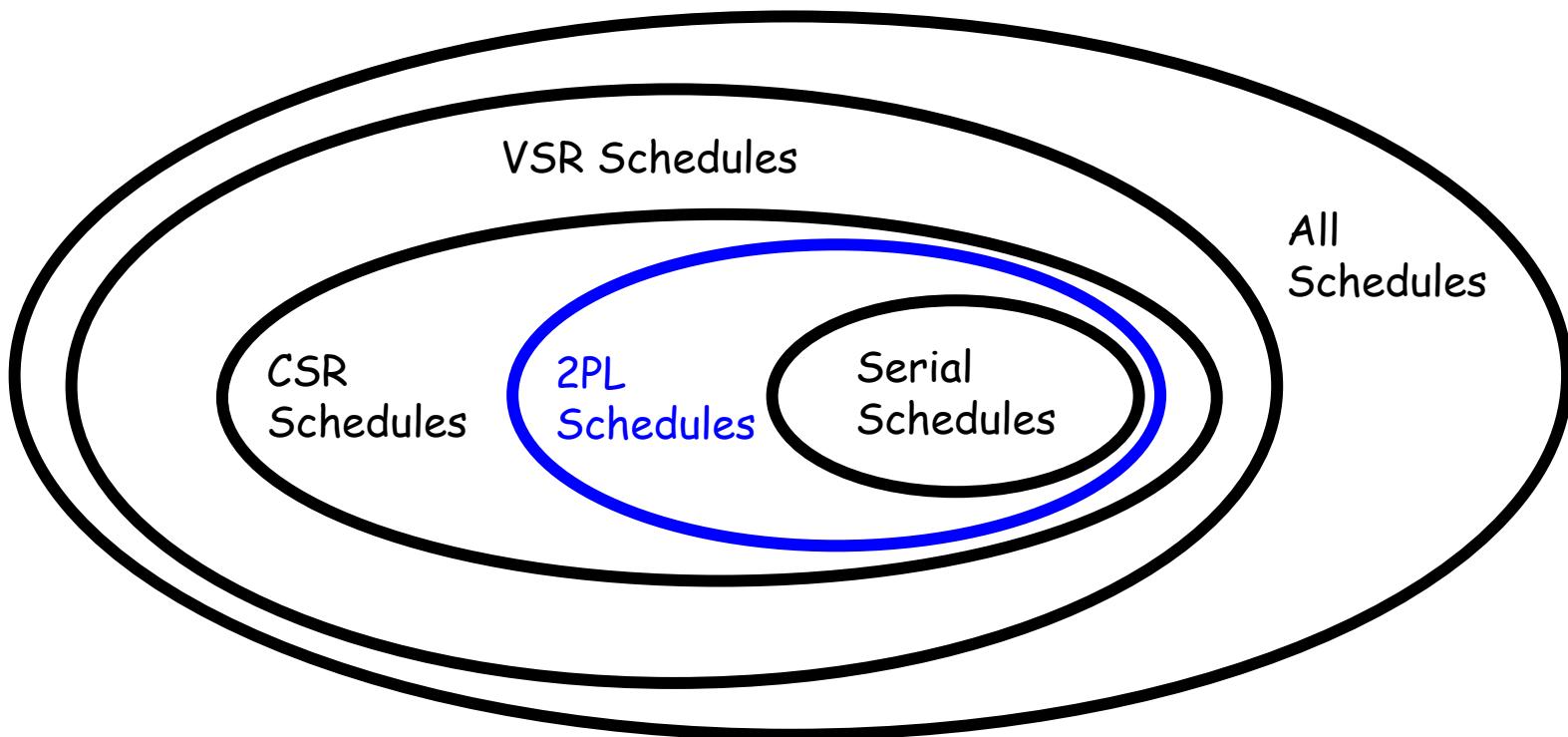
Resources on the Y axis and operation times on the X axis



2PL and other anomalies

- Nonrepeatable read r1 - r2 - w2 - r1
 - Already shown
- Lost update r1 - r2 - w2 - w1
 - T1 releases a lock to T2 and then tries to acquire another one
- Phantom update: r1(A,B) - r2(B,C) - w2(BC) - r1(C)
 - T1 releases a lock to T2 and then tries to acquire another one
- Phantom insert: r1 - w2(new data) - r1
 - T1 releases a lock to T2 and then tries to acquire another one
(NOTE: T2 does not necessarily write on data already locked by T1 → requires lock on “future data” aka predicate locks)
- Dirty read r1 - w1 - r2 - abort1 - w2
 - Requires dealing with abort

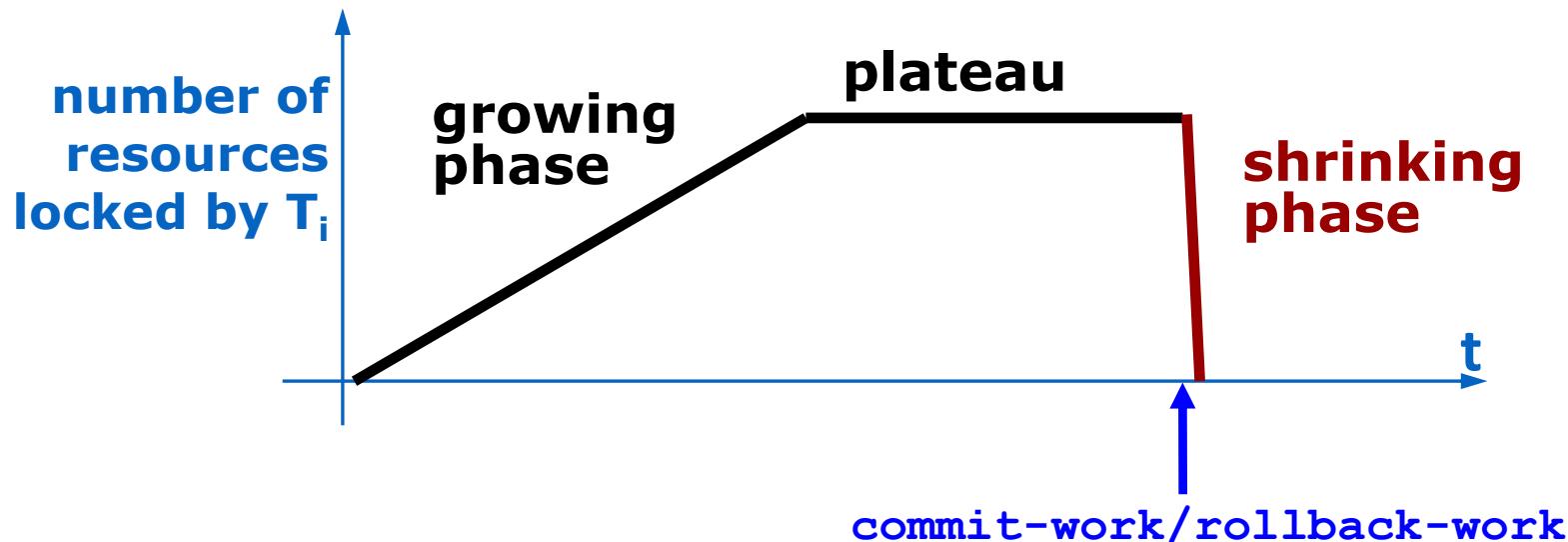
CSR, VSR and 2PL



Dirty reads are still a menace: Strict 2PL

- Up to now, we were still using the hypothesis of commit-projection (no transactions in the schedule abort)
 - 2PL, as seen so far, does not protect against dirty (uncommitted data) reads (and therefore neither do VSR nor CSR)
 - Releasing locks before rollbacks exposes “dirty” data
- To remove this hypothesis, we need to add a constraint to 2PL, that defines **strict 2PL**:
 - Locks held by a transaction can be released only after commit/rollback
 - Remember: rollback restores the state prior to the aborted updates
- This version of 2PL is used in most commercial DBMSs when a high level of isolation is required (see next: SQL isolation levels)

Strict 2PL in Practice



- Strict 2PL locks are also called **long duration** locks, 2PL locks **short duration** locks
- Note: real systems may apply 2PL policies differently to read and write locks
- Typically: long duration strict 2PL write locks, variable policies for read locks
- NOTE: long duration read locks are costly in terms of performances: real systems replace them with more complex mechanisms

How to prevent phantom inserts: predicate locks

- A phantom insertion occurs when a transaction adds items to a data set previously read by another transaction
- To prevent phantom inserts a lock should be placed also on “future data”, i.e., inserted data that **would satisfy** a previous query
- **Predicate locks** extend the notion of **data locks** to “future data”
- A transaction T = update Tab set B=1 where A<1
- Then, the lock is on predicate A<1
 - Other transactions cannot insert, delete, or update any tuple satisfying this predicate
 - In the worst case (predicate locks not supported):
 - The lock extends to the entire table
 - In case the implementation supports predicate locks:
 - The lock is managed with the help of indexes (gap lock)

Tab	
A	B

Isolation Levels in SQL:1999 (and JDBC)

- SQL defines transaction isolation levels which specify the anomalies that should be prevented by running at that level
- The level **does not affect write locks**. A transaction should always get exclusive lock on any data it modifies, and hold it until completion (**strict 2P on write locks**), regardless of the isolation level. For read operations, levels define the degree of protection from the effects of modifications made by other transactions

Why long duration write locks are necessary

- Consider the following schedule (admissible if write locks are short duration) and **remove the hypothesis that aborts do not occur**
- $w_1[x] \dots w_2[x] \dots ((c_1 \text{ or } a_1) \text{ and } (c_2 \text{ or } a_2))$ in any order
 - T2 is allowed to write over the same object updated by T1 which has not yet completed but unlocked x
- If T1 aborts... e.g., $w_1[x] \dots w_2[x] \dots , a_1, (c_2 \text{ or } a_2)$
 - How to process event a1?
 - If x is restored to the state before T1, T2's update is lost, so if T2 commits x has a stale value
 - If x is NOT restored and T2 also aborts... then T2's proper before state cannot be reinstalled either!
- Thus: write locks are held until the completion of the transaction to enable the proper processing of abort events
- The anomaly of the above non commit-projection schedule is named **dirty write**

Isolation Levels in SQL:1999 (and JDBC)

- READ UNCOMMITTED allows dirty reads, nonrepeatable reads and phantom updates and inserts:
 - No read locks (and **ignores** locks of other transactions)
- READ COMMITTED prevents dirty reads but allows nonrepeatable reads and phantom updates/inserts:
 - Read locks (and complies with locks of other transactions), but without 2PL on read locks (read locks are released as soon as the read operation is performed and can be acquired again)
- REPEATABLE READ avoids dirty reads, nonrepeatable reads and phantom updates, but allows phantom inserts:
 - long duration read locks → strict 2PL also for reads
- SERIALIZABLE avoids all anomalies:
 - Strict 2PL with predicate locks to avoid phantom inserts
- Note that SQL standard isolation levels dictate minimum requirements, real systems may go beyond (e.g., in MySQL and Postgres REPEATABLE READ avoids phantom inserts too) and use different mechanisms (e.g., to avoid long duration read locks)

	Dirty Read	Non rep. read	Phantoms
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y (insert)
Serializable	N	N	N

SQL92 serializable <> serial !

- Serializable transactions don't necessarily execute serially
- The requirement is that transactions can only commit if the result would be as if they had executed serially in any order
- The locking requirements to meet this guarantee can frequently lead to a **deadlock** where one of the transactions needs to be rolled back
- Therefore, the **SERIALIZABLE** isolation level is used **sparingly** and is NOT the default in most commercial systems

SQL isolation levels and locks

- SQL Isolation levels may be implemented with the appropriate use of locks
- Commercial systems make joint use of locks and of timestamp-based concurrency control mechanisms

	READ LOCKS	WRITE LOCKS
READ UNCOMMITTED	Not required	Well formed writes Long duration write locks
READ COMMITTED	Well formed reads Short duration read locks (data and predicate)	Well formed writes Long duration write locks
REPEATABLE READ	Well formed reads Long duration data read locks Short duration predicate read locks	Well formed writes Long duration write locks
SERIALIZABLE	Well formed reads Long duration read locks (predicate and data)	Well formed writes Long duration write locks

Setting transaction characteristics in SQL

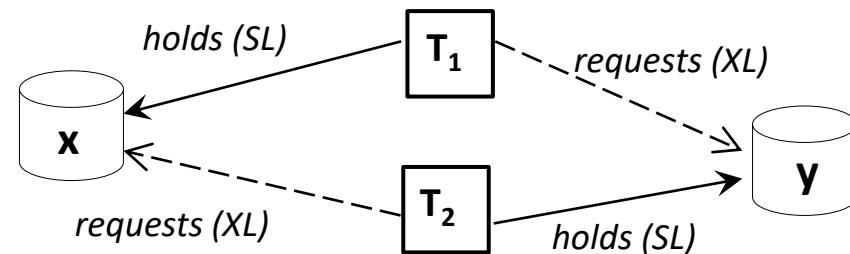
```
<set transaction statement> ::=  
SET [ LOCAL ] TRANSACTION <transaction characteristics>  
  
<transaction characteristics> ::= [ <transaction mode> [ { <comma>  
    <transaction mode> }... ] ]  
  
<transaction mode> ::= <isolation level> | <transaction access  
mode>  
| <diagnostics size>  
  
<transaction access mode> ::= READ ONLY | READ WRITE  
  
<isolation level> ::= ISOLATION LEVEL <level of isolation>  
  
<level of isolation> ::= READ UNCOMMITTED | READ COMMITTED  
| REPEATABLE READ | SERIALIZABLE
```

The impact of locking: waiting is dangerous!

- Locks are tracked by lock tables (main memory data structures)
 - Resources can be Free, or Read-Locked, or Write-locked
 - To keep track of readers, every resource also has a "read counter"
- Transactions requesting locks are either **granted the lock** or **suspended and queued** (first-in first-out). There is risk of:
 - **Deadlock:** two or more transactions in endless (mutual) wait
 - Typically occurs when each transaction waits for another to release a lock (in particular: r1(x) r2(y) w1(y) w2(x) → see update lock later)
 - **Starvation:** a single transaction in endless wait
 - Typically occurs due to write transactions waiting for resources that are continuously read (e.g., index roots)

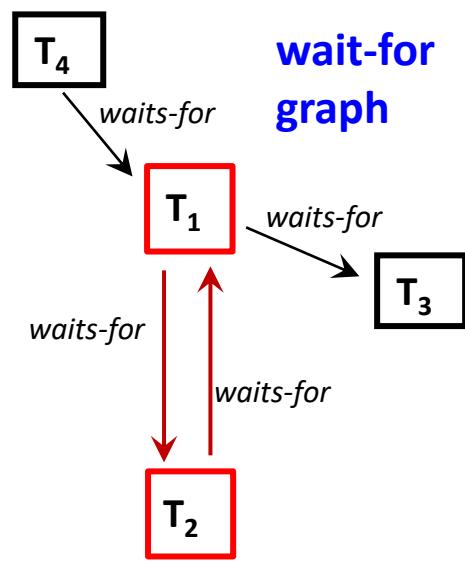
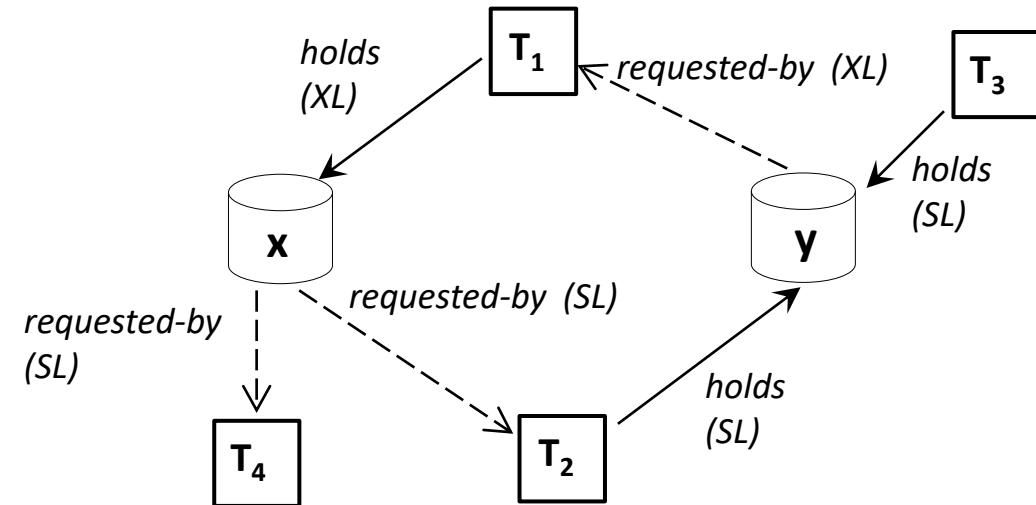
Deadlock

- Occurs because concurrent transactions hold and in turn request resources held by other transactions
- T1 : r1(x) w1(y)
- T2 : r2(y) w2(x)
- S: **r_lock1(x), r_lock2(y), r1(x), r2(y), w_lock1(y), w_lock2(x) → deadlock**



Deadlock

- **Lock graph:** a bipartite graph in which nodes are resources or transactions and arcs are lock requests or lock assignments
- **Wait-for graph:** a graph in which nodes are transactions and arcs are “waits for” relationships
- A deadlock is represented by a cycle in the wait-for graph of transactions



Deadlock Resolution Techniques

- **Timeout**
 - Transactions killed after a long wait
 - How long?
- **Deadlock prevention**
 - Transactions killed when they COULD BE in a deadlock
 - Heuristics
- **Deadlock detection**
 - Transactions killed when they ARE in a deadlock
 - Inspection of the wait-for graph

Deadlock detection in commercial dashboards

The screenshot displays a monitoring interface with the following sections:

- Top Bar:** Includes filters for "All groups" (6), "All Base monitors" (4), and "Severity" (2). A search bar and alert count (24) are also present.
- Left Sidebar:** Shows navigation paths: "1 - PRODUCTION" and "2 - AZURE DATA...".
- Alert Summary:** A message states: "You can now monitor Amazon RDS SQL Servers. [Learn more](#) or send [feedback](#)."
- Status at a Glance:** A central card provides an overview of server health. It shows three servers: "sqlservercentral_test" (orange, Deadlock event), "monitor-elasticpool" (green, healthy), and "sqm-sqlmonitor\sqlmonitor" (yellow, Database file usage). Below this, a "STATUS AT A GLANCE" section describes how cards rearrange to highlight issues and show metrics like %PU, DATA I/O, and LOG I/O.
- Performance Metrics:** Three cards show real-time performance data:
 - "sqm-sqlmonitor\sqlmonitor": Database file usage, monitor.red-gate.com since ...
Metrics: 11ms/s WAITS, 7% CPU, 82kB/s DISK I/O.
 - "SQL Server Reporting Service status (2017+)" since ...
Metrics: 163ms/s WAITS, 7% CPU, 13MB/s DISK I/O.
 - "SQL Server Reporting Service status (2017+)" since ...
Metrics: 8s/s WAITS, 1% CPU, 57kB/s DISK I/O.
- Right Panel:** A list of active alerts grouped by severity:
 - Monitoring stopped (host) (7 active alerts)
 - Deadlock (extended event) (1 active alert)
 - Database file usage (1 active alert)
 - Blocking process (15 active alerts)A "See all alerts" button is located at the bottom.

Timeout Method

- A transaction is killed and restarted after a given amount of waiting (assumed as due to a deadlock)
 - The simplest method, widely used in the past
- The timeout value is system-determined (sometimes it can be altered by the database administrator)
- The problem is choosing a proper timeout value
 - Too long: useless waits whenever deadlocks occur
 - Too short: unrequired kills, redo overhead
- <http://daveblanchard.com/how-often-does-sql-server-look-for-deadlocks>

Deadlock Prevention

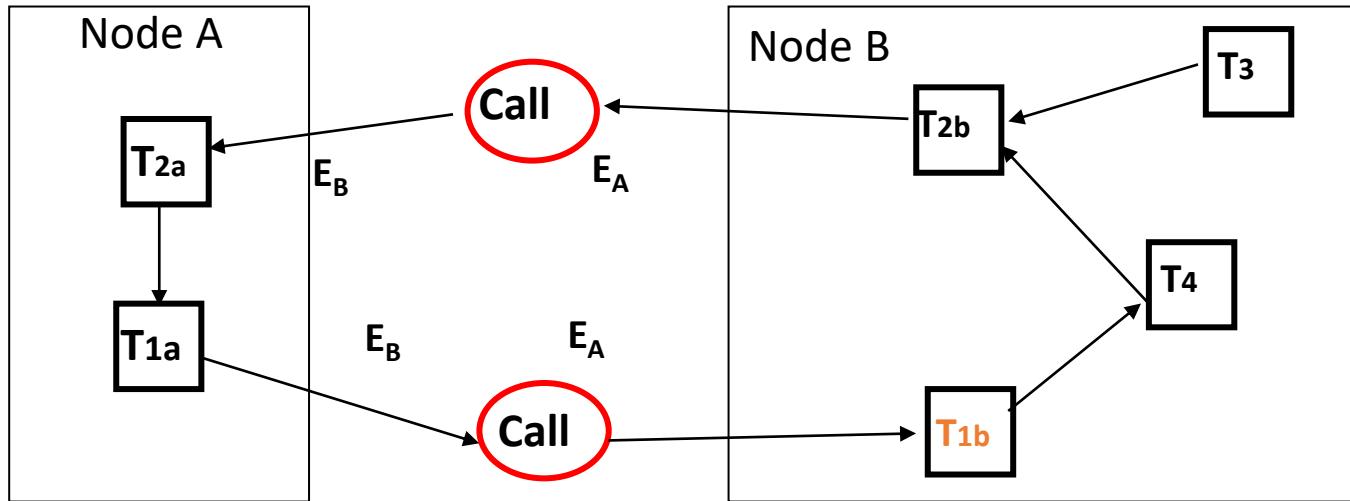
- Idea: killing transactions that could cause cycles
- **Resource-based prevention:** Restrictions on lock requests
 - Transactions request all resources at once, and only once
 - Resources are globally sorted and must be requested “in global order”
 - Problem: it’s not easy for transactions to anticipate all requests!
- **Transaction-based prevention:** restrictions based on transactions’ IDs
 - Assigning IDs to transactions incrementally → transactions’ “age”
 - Preventing “older” transactions from waiting for “younger” ones to end their work
 - Options for choosing the transaction to kill
 - **Preemptive** (killing the holding transaction – wound-wait)
 - **Non-preemptive** (killing the requesting transaction – wait-die)
 - Problem: too many “killings”! (waiting probability >> deadlock probability)

Deadlock Detection

- Requires an algorithm to detect cycles in the wait-for graph
 - Must work with **distributed** resources efficiently & reliably
- An elegant solution: **Obermarck's** algorithm (DB2-IBM, published on ACM Transactions on Database Systems)
- Assumptions
 - Transactions execute on a single main node (one locus of control)
 - Transactions may be decomposed in “sub-transactions” running on other nodes
 - Synchronicity: when a transaction spawns a sub-transaction it suspends work until the latter completes
 - Two wait-for relationships:
 - T_i waits for T_j on the same node because T_i needs a datum locked by T_j
 - A sub-transaction of T_i waits for another sub-transaction of T_i running on a different node (via external call E)

Distributed Deadlock Detection

Distributed dependency graph: external call nodes represent a sub-transaction activating another sub-transaction at a different node

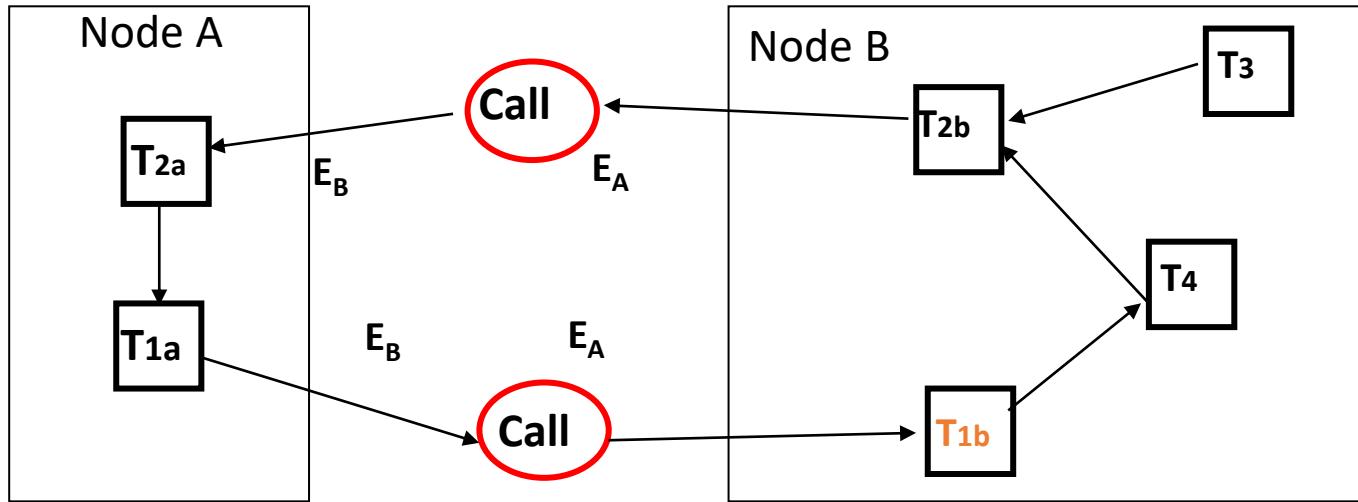


- Representation of the status
 - at Node A: E_b → T_{2a} → T_{1a} → E_b
 - at Node B: E_a → T_{1b} → T_{2b} → E_a
- NOTATION: symbol → : 1) “wait for” relation among local transactions 2) if one term is an external call, either the source is waited for by a remote transaction or the sink waits for a remote transaction
- Potential Deadlock: T_{2a} waits for T_{1a} (data lock) that waits for T_{1b} (call) that waits for T_{2b} (data locks) that waits for T_{2a} (call): cycle between T1 and T2!
- Problem: how to detect such an occurrence without maintaining the global view

Obermarck's Algorithm

- Goal: detection of a potential deadlock looking only at the local view of a node
- Method: establishing a communication protocol whereby each node has a local projection of the global dependencies
- Nodes exchange information and update their local graph based on the received information
- Communication is optimized to avoid that multiple nodes detect the same potential deadlock
- Node A sends its local info to a node B only if
 - A contains a transaction T_i that is waited for from another remote transaction and waits for a transaction T_j active on B
 - $i > j$ (this ensure a kind of message “forwarding” along a node path where node A “precedes” node B if $i > j$)
 - Mnemonically: I send info to you if a distributed transaction listed at me waits for a distributed transaction listed at you with **smaller** index

Forwarding rule

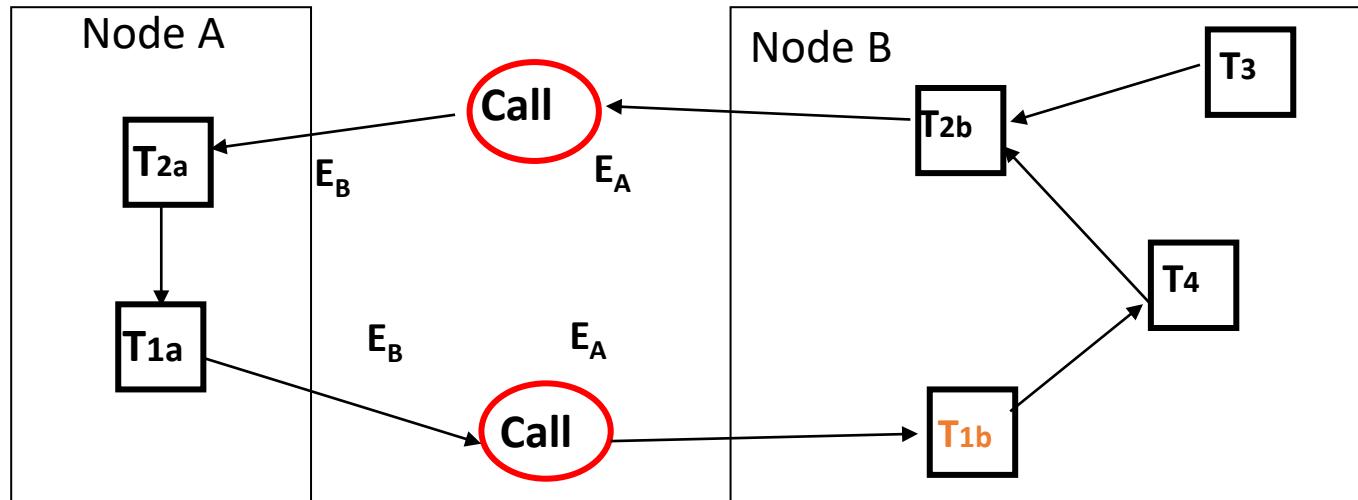


- **Node A:**
 - Activation/wait sequence: E_b → T₂ → T₁ → E_b
 - i=2, j=1
 - A can dispatch info to B
- **Node B:**
 - Activation/wait sequence (only distributed transactions count): E_a → T₁ → T₂ → E_a
 - i=1, j=2
 - B does not dispatch info to A

Obermarck's Algorithm

- Runs periodically at each node
- Consists of 4 steps
 - Get graph info (wait dependencies among transactions and external calls) from the “previous” nodes. Sequences contain only node and top-level transaction identifiers
 - Update the local graph by merging the received information
 - Check the existence of cycles among transactions denoting potential deadlocks: if found, select one transaction in the cycle and kill it
 - Send updated graph info to the “next” nodes

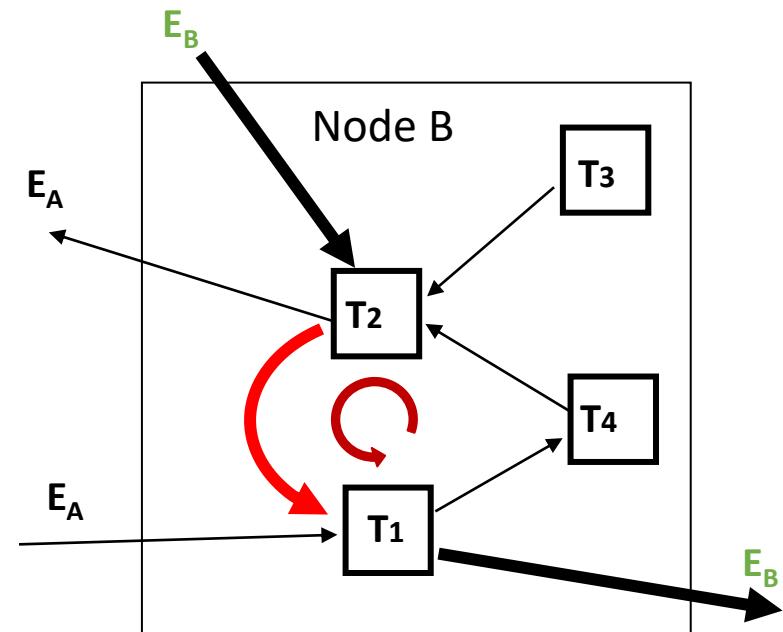
Algorithm execution, step 1: communication



- Distributed Deadlock Detection: forwarding rule
 - at Node A: $E_b \rightarrow T_2 \rightarrow T_1 \rightarrow E_b$ info sent to Node B
 - at Node B: $E_a \rightarrow T_1 \rightarrow T_2 \rightarrow E_a$ info not sent ($i < j$)

Algorithm execution, step 2: local graph update

- at Node B:
- 1. $E_B \rightarrow T_2 \rightarrow T_1 \rightarrow E_B$ is received
- 2. $E_B \rightarrow T_2 \rightarrow T_1 \rightarrow E_B$ is added to the local wait-for graph
- 3. Deadlock detected (cycle among T_1 and T_2):
- T_1 or T_2 or T_4 killed (rollback)

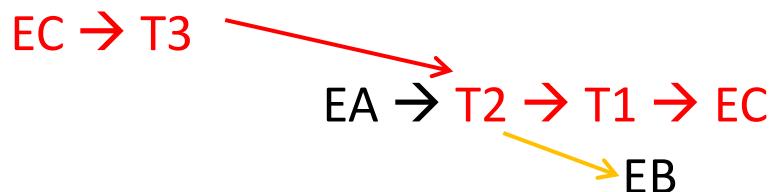


Another example

- Initially: at Node A, EC → T3 → T2 → EB
at Node B, EA → T2 → T1 → EC
at Node C, EB → T1 → T3 → EA
- Node A: 3 > 2 can send (to B)
- Node B: 2 > 1 can send (to C)
- Node C: 1 < 3 cannot send
- We assume that node A is the first to execute, sending the sequence «EC → T3 → T2 → EB » to B (i.e., to the node hosting the (sub)transaction that keeps A's transaction on hold)

Another example, continued

- At node B:
- $EA \rightarrow T2 \rightarrow T1 \rightarrow EC$ is merged with $EC \rightarrow T3 \rightarrow T2 \rightarrow EB$
- Result:



$T3$ is waited for by a remote call
and waits for $T1$ which waits for a
remote call

- A new potential deadlock $EC \rightarrow T3 \rightarrow T1 \rightarrow EC$ is found by combining the $EC \rightarrow T3 \rightarrow T2 \rightarrow EB$ message with the local info $EA \rightarrow T2 \rightarrow T1 \rightarrow EC$. This can be sent to node C (3>1)
- At node C:
- Local info $EB \rightarrow T1 \rightarrow T3 \rightarrow EA$ is merged with message $EC \rightarrow T3 \rightarrow T1 \rightarrow EC$
 $EB \rightarrow T1 \leftarrow \rightarrow T3 \rightarrow EA$
- The deadlock is found and either $T1$ or $T3$ is killed.

Obermarck: immateriality of conventions

- There are two arbitrary choices in the algorithm:
- Send messages only if: (1) $i > j$ vs. (2) $i < j$
- Send them to: (a) the following node vs. (b) the preceding node
- Therefore, there are four versions/variants of the algorithm
 - (1+a), (1+b), (2+a), (2+b)
 - The sequence of the sent messages is different
 - However, they all identify deadlocks (if present)

Deadlocks in practice

- Their probability is much less than the conflict probability
 - Consider a file with n records and two transactions doing two accesses to their records (uniform distribution); then:
 - Conflict probability is $O(1/n)$: $\sum_{i=1..n} (1/n * 1/n) = n * (1/n * 1/n) = 1/n$
 - Deadlock probability is $O(1/n^2)$: T_1 conflicts with T_2 AND vice versa
- Still, they do occur (once every minute in a mid-size bank)
 - The probability is linear in the number of transactions, **quadratic in their length** (measured by the number of lock requests)
 - Shorter transactions are healthier (*ceteris paribus*)
- There are techniques to limit the frequency of deadlocks
 - Update Lock, Hierarchical Lock, ...,

Update lock

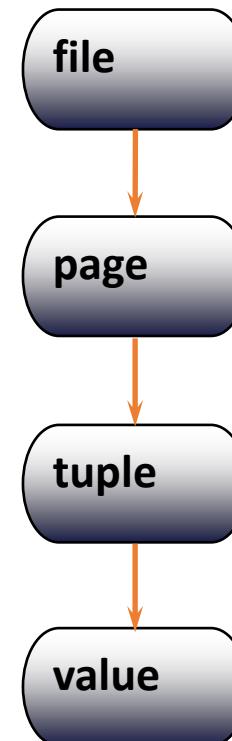
- The most frequent deadlock occurs when 2 concurrent transactions start by reading the same resources (SL) and then decide to write and try to upgrade their lock to XL
- To avoid this situation, systems offer the UPDATE LOCK (UL)
 - asked by transactions that will **read and then write**

		Resource status		
Request		free	SL	UL
SL	free	OK	OK	OK
	UL	OK	OK	No
XL	OK	No	No	No

- Update locks are easy to implement and mitigate the most frequent cause of collision: $r1(x)$ $r2(x)$ $w1(x)$ $w2(x)$
- They are requested by using SQL SELECT FOR UPDATE statement

Hierarchical Locking

- Update locks prudentially extend the interval during which a resource is locked
- What to lock? An entire table? → reduces concurrency too much
- Locks can be specified with different granularities
 - e.g.: schema, table, fragment, page, tuple, field
- Objectives:
 - Locking the minimum amount of data
 - Recognizing conflicts as soon as possible
- Method: asking locks on hierarchical resources by:
 - Requesting resources top-down until the right level is obtained
 - Releasing locks bottom-up



**Coarser
Granularity**

**Increased
Concurrency**

Intention Locking Scheme

- 5 Lock modes:
 - In addition to read (SHARED) locks (SL) and write (EXCLUSIVE) locks (XL)
- The new modes express the “**intention**” of locking at lower (finer) levels of granularity
 - **ISL**: Intention of locking a subelement of the current element in shared mode
 - **IXL**: Intention of locking a subelement of the current element in exclusive mode
 - **SIXL**: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)

Hierarchical Locking Protocol

- Locks are requested starting from the root (e.g., starting from the whole table) and going down in the hierarchy
- Locks are released starting from the leaves and going up in the hierarchy
- To request an SL or ISL lock on a non-root element, a transaction must hold an equally or more restrictive lock (ISL or IXL) on its “parent”
- To request an IXL, XL or SIXL lock on a non-root element, a transaction must hold an equally or more restrictive lock (SIXL or IXL) on its “parent”
- When a lock is requested on a resource, the lock manager decides based on the rules specified in the hierarchical lock granting table

Hierarchical lock granting table

T_x may share lock the whole table in only one shot and then update some rows (e.g., set one employee's salary to the average value of the salary of the colleagues of the same unit)

T_y may set an ISL lock on the table then read some other rows of the same table

Without SIXL, T_x should use IXL on the table and then acquire multiple locks on the rows (lock overhead)

Request	Resource state					
	free	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	OK	No
IXL	OK	OK	OK	No	No	No
SL	OK	OK	No	OK	No	No
SIXL	OK	OK	No	No	No	No
XL	OK	No	No	No	No	No

Example

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

Root = TableX

Page 1 (P1): t1,t2,t3,t4 tuples

Page 2 (P2): t5,t6,t7,t8 tuples

Transaction 1: `read(P1)`

`write(t3)`

`read(t8)`

Transaction 2: `read(t2)`

`read(t4)`

`write(t5)`

`write(t6)`

They are NOT in r-w conflict!
(independently of the order)

Lock Sequences

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

P1	P2
t1	t5
t2	t6
t3	t7
t4	t8

Transaction 1: **read(P1)**
write(t3)
read(t8)

T 1:

IXL(root)
SIXL(P1)
XL(t3)
ISL(P2)
SL(t8)

Transaction 2: **read(t2)**
read(t4)
write(t5)
write(t6)

T2:

IXL(root)
ISL(P1)
SL(t2)
SL(t4)
IXL(P2)
XL(t5)
XL(t6)

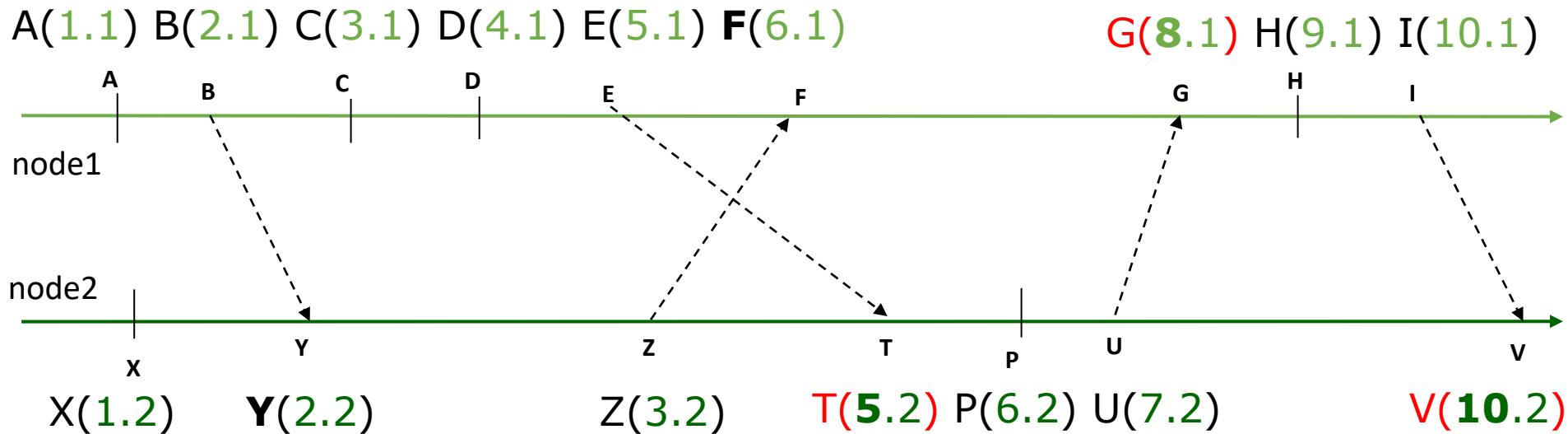
Concurrency Control Based on Timestamps

- Locking is also named **pessimistic** concurrency control because it assumes that collisions (transactions reading-writing the same object concurrently) will arise
- In reality collisions are rare
- Alternative and complementary to 2PL (and to locking in general) are **optimistic** concurrency control methods
- **Timestamp:**
 - Identifier that defines a total ordering of the events of a system
 - Each transaction has a timestamp representing the time at which the transaction begins so that transactions can be ordered by “birth date”: smaller index → older transaction
 - A schedule is accepted only if it reflects the serial ordering of the transactions induced by their timestamps

Assigning timestamps in distributed systems

- Timestamp: an indicator of the “current time”
- Assumption: no “global time” is available
- Mechanism: a system’s function gives out timestamps on requests
- Syntax: timestamp = **event-id.node-id**
 - event-ids are unique at each node
- Note that the notion of time is “lexical”: timestamp **5.1** “occurs before” timestamp **5.2**
- Synchronization: send-receive of messages
 - for a given message m, send(m) precedes receive(m)
- Algorithm (Lamport method): cannot receive a message from “the future”, if this happens the “bumping rule” is used to bump the timestamp of the receive event beyond the timestamp of the send event
- **Mnemonically:** if I receive a message from you that has a timestamp greater than my last emitted one I update my current timestamp to exceed yours

Example of timestamp assignment



- Events Y and F represent messages received “from the present or past” → OK the local timestamp is aligned without bumping
- Events T, G and V represent messages received “from the future” → The local timestamp is incremented (bumped) accordingly leaving a “hole” in the local event sequence
- Note that the timestamp of the receive events T, G and V is generated so as to exceed that of the send event: G(8.1)>U(7.2)

TS concurrency control principles

smaller index → older transaction

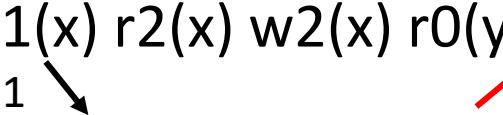
- The scheduler has two counters: $\text{RTM}(x)$, $\text{WTM}(x)$ for each object: index of last reader, writer
- The scheduler receives read/write requests tagged with the timestamp of the requesting transaction:
 - $r_{ts}(x)$: --the requesting reader is older than the one who last wrote
 - If $ts < \text{WTM}(x)$ the request is **rejected** and the transaction is killed
 - Else, access is **granted** and $\text{RTM}(x)$ is set to $\max(\text{RTM}(x), ts)$
 - $w_{ts}(x)$: --the requesting writer is older than the one who last read or wrote
 - If $ts < \text{RTM}(x)$ or $ts < \text{WTM}(x)$ the request is **rejected** and the transaction is killed
 - Else, access is **granted** and $\text{WTM}(x)$ is set to ts
- Many transactions are killed

Example

Assume $\text{RTM}(x) = 7$ -- last read by T7
 $\text{WTM}(x) = 4$ -- last written by T4

Request	Response	New value
$r_6(x)$	ok	
$r_8(x)$	ok	$\text{RTM}(x) = 8$
$r_9(x)$	ok	$\text{RTM}(x) = 9$
$w_8(x)$	no	T8 killed, x already read by T9
$w_{11}(x)$	ok	$\text{WTM}(x) = 11$
$r_{10}(x)$	no	T10 killed, x already written by T11

2PL vs. TS

- They are incomparable
- Schedule in TS but not in 2PL
 - $r1(x) w1(x) r2(x) w2(x) r0(y) w1(y)$

- Schedule in 2PL but not in TS
 - $r2(x) w2(x) r1(x) w1(x)$
- Schedule both in TS and in 2PL
 - $r1(x) r2(y) w2(y) w1(x) r2(x) w2(x)$

- NOTE: $r2(x) w2(x) r1(x) w1(x)$ is **serial but not in TS**

TS and CSR

- **TS => CSR**
 - Let S be a TS schedule of T1 and T2
 - Suppose S is not CSR, which implies that it contains a cycle between T1 and T2
 - S contains $op_1(x)$, $op_2(x)$ where at least one of the op_i is a write
 - S contains also $op_2(y)$, $op_1(y)$ where at least one of the op_i is a write
 - When $op_1(y)$ arrives:
 - If $op_1(y)$ is a read, T1 is killed by TS because it tries to read a value written by a younger transaction [$ts < WTM(y)$] [$1 < 2$] → CONTRADICTION
 - If $op_1(y)$ is a write, T1 is killed no matter what $op_2(y)$ is because it tries to write a value already read or written by a younger transaction [$ts < R/WTM(y)$] [$1 < 2$] → CONTRADICTION

TS and dirty reads

- Basic TS-based control considers only committed transactions in the schedule, aborted transactions are not considered (commit-projection hypothesis)
- If aborts occur, dirty reads may happen
- To cope with dirty reads, a variant of basic TS must be used
 - A transaction T_i that issues a $r_{ts}(x)$ or $w_{ts}(x)$ such that $ts > \text{WTM}(x)$ (i.e., acceptable) has its read or write operation **delayed** until the **transaction T' that wrote the value of x has committed or aborted**
 - Similar to long duration write locks
 - But...buffering operations introduces delays

CSR, VSR, 2PL and TS

X : r₁ w₁ r₂ w₂

All schedules

VSR

CSR

TS

2PL

Serial

X : r₂ w₂ r₁ w₁



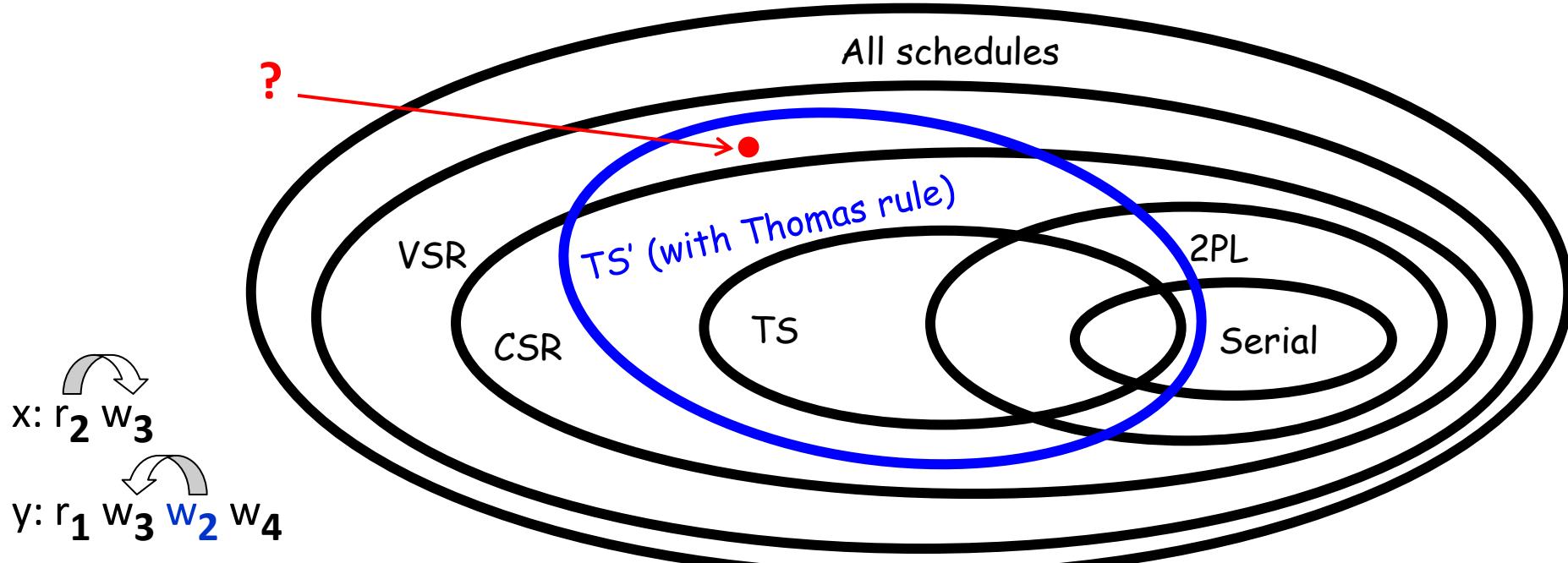
2PL vs. TS

- In 2PL transactions can be actively waiting. In TS they are killed and restarted
- The serialization order with 2PL is imposed by conflicts, while in TS it is imposed by the timestamps
- The necessity of waiting for commit of transactions causes long delays in strict 2PL
- 2PL can cause deadlocks, TS can be used to prevent deadlocks with the wound-wait and wait-die schemes explained before
- Restarting a transaction costs more than waiting: 2PL wins!
- Commercial systems implement a mix of optimistic and pessimistic concurrency control (e.g., Strict 2PL or 2PL + Multi Version TS)

Reducing kill rate: Thomas Rule

- The scheduler has two counters: $RTM(x)$ and $WTM(x)$ for each object
- The scheduler receives read/write requests tagged with timestamps:
 - $r_{ts}(x)$:
 - If $ts < WTM(x)$ the request is rejected and the transaction is killed
 - Else, access is granted and $RTM(x)$ is set to $\max(RTM(x), ts)$
 - $w_{ts}(x)$:
 - If $ts < RTM(x)$ the request is **rejected** and the transaction is killed
 - Else, if $ts < WTM(x)$ then our write is "obsolete": it can be **skipped**
 - Else, access is **granted** and $WTM(x)$ is set to ts
- Rationale: skipping a write on an object that has already been written by a younger transaction, without killing the transaction
- Works only if the transaction issues a write without requiring a previous read on the object (so $SET X = X+1$ would fail)
- Does this modification affect the taxonomy of the serialization classes?

TS' (TS with Thomas Rule)

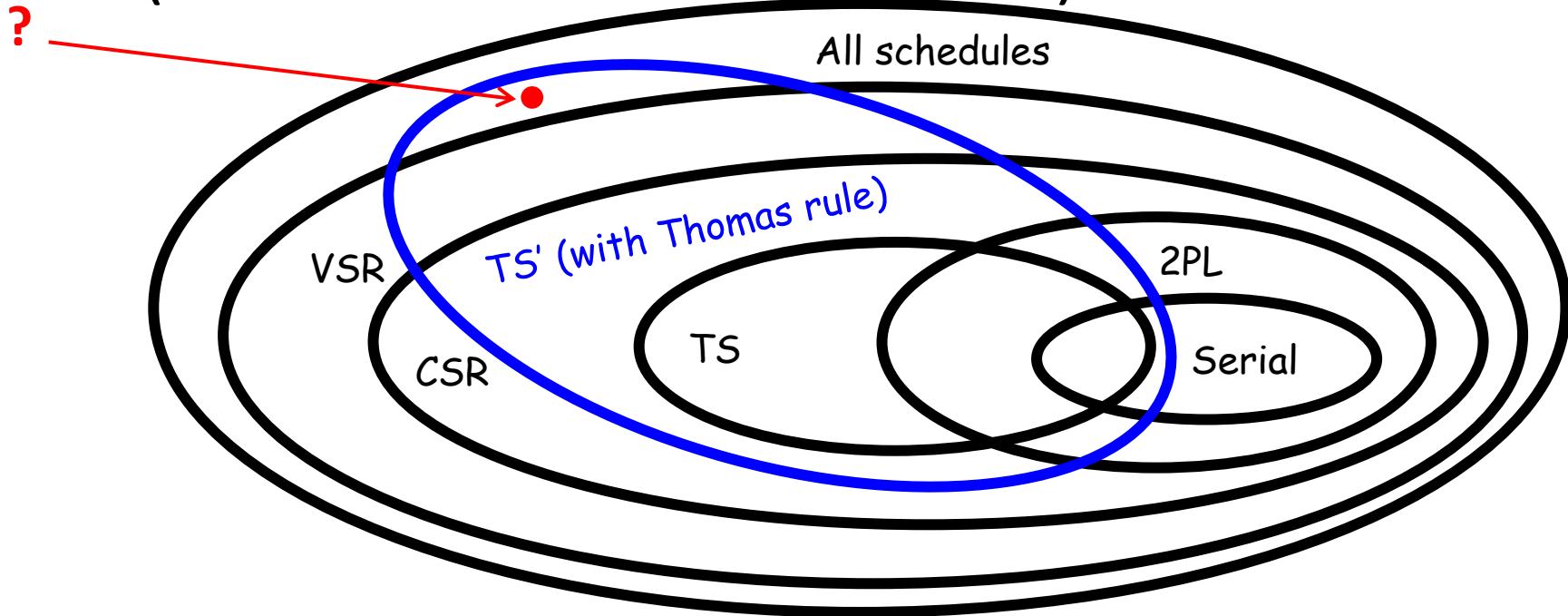


$r_1(y) r_2(x) w_3(y) \textcolor{blue}{w_2(y)} w_3(x) w_4(y)$

$\textcolor{blue}{w_2(y)}$ is an obsolete write and thus skipped

- This arrival sequence is accepted by Thomas Rule, but we are changing the semantics, since not all of it is executed

TS' (TS with Thomas Rule)



- $w_2(x) \text{ } w_1(x) \text{ } r_2(x)$
 - $w_1(x)$ skipped by Thomas Rule
- Schedule not in VSR
 - T1, T2 different final write relation
 - T2, T1 different reads-from relation for $r_2(x)$

Multiversion Concurrency Control

- Idea: **writes generate new versions, reads access the "right" version**
- Writes generate new copies, each one with a new WTM. Each object x always has $N \geq 1$ active versions
 - The i -th version of x has its write timestamp $WTM_i(x)$
- There is a unique object $RTM(x)$
- Old versions are discarded when there are no transactions that need their values

1st version (used in theory): TS-Multi allowing unordered writes

- Mechanism:

- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading such that:
 - If $ts \geq WTM_N(x)$, then $k = N$
 - Else take k such that $WTM_k(x) \leq ts < WTM_{k+1}(x)$
- $w_{ts}(x)$:
 - If $ts < RTM(x)$ the request is **rejected**
 - Else a new version is **created** for timestamp ts (N is incremented)
 - $WTM_1(x), \dots, WTM_N(x)$ are the new versions, kept sorted from oldest to youngest
 - NB: this version shows what can be done in theory but is not the one used in the exercises

Example in theory:

TS-Multi allowing unordered writes

Assume $\text{RTM}(x) = 7$, $N=1$, $\text{WTM}_1(x) = 4$

<i>Request</i>	<i>Response</i>	<i>New Value</i>
$r_6(x)$	ok	
$r_8(x)$	ok	$\text{RTM}(x) = 8$
$r_9(x)$	ok	$\text{RTM}(x) = 9$
$w_8(x)$	no	T_8 killed
$w_{11}(x)$	ok	$\text{WTM}_2(x) = 11$, $N=2$
$r_{10}(x)$	ok on x_1	$\text{RTM}(x) = 10$ (no longer killed)
$r_{12}(x)$	ok on x_2	$\text{RTM}(x) = 12$
$w_{14}(x)$	ok	$\text{WTM}_3(x) = 14$, $N=3$
$w_{13}(x)$	ok	$\text{WTM}_1(x)=4$, $\text{WTM}_2(x)=11$, $\text{WTM}_3(x)=13$, $\text{WTM}_4(x)=14$, $N=4$ (no need to kill)

2nd version (used in practice): TS-Multi under Snapshot Isolation

- Mechanism:

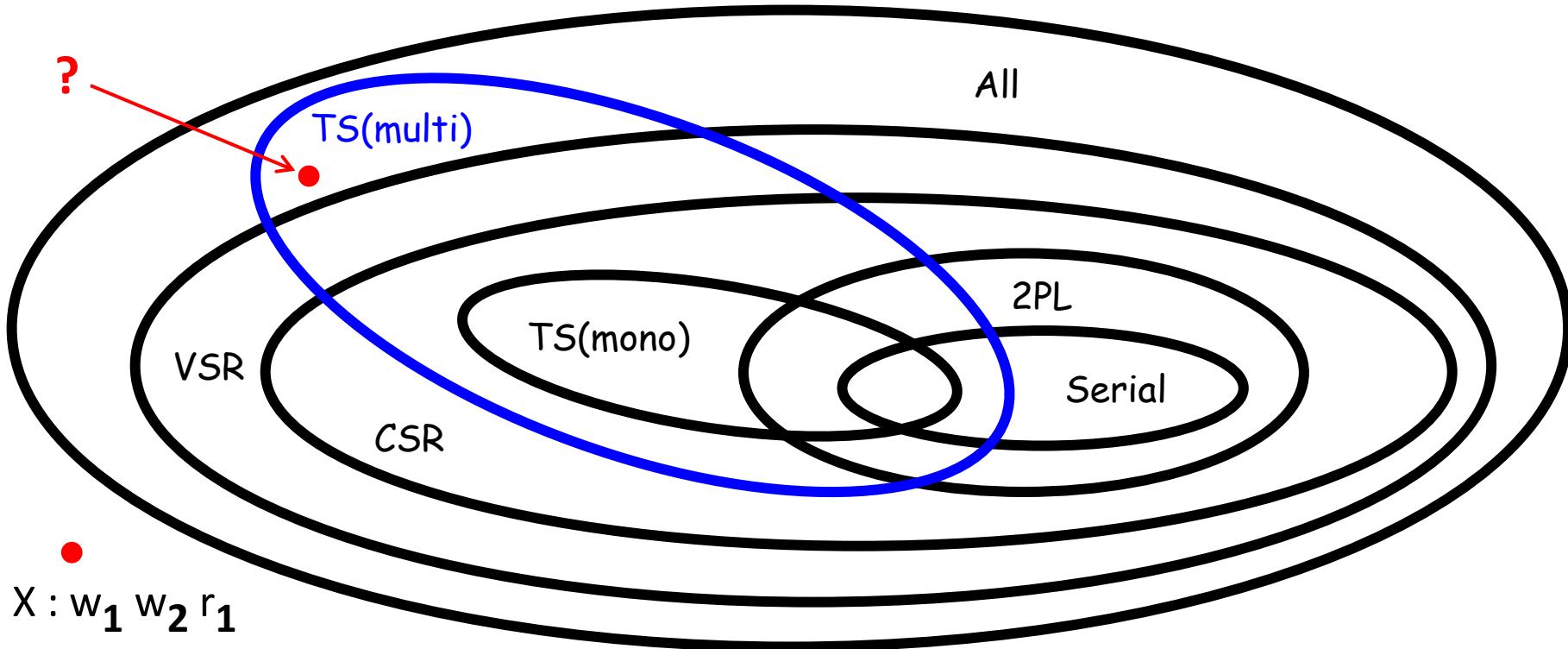
- $r_{ts}(x)$ is always accepted. A copy x_k is selected for reading such that:
 - If $ts \geq WTM_N(x)$, then $k = N$
 - Else take k such that $WTM_k(x) \leq ts < WTM_{k+1}(x)$
- $w_{ts}(x)$:
 - If $ts < RTM(x)$ or $ts < WTM_N(x)$ the request is **rejected**
 - Else a new version is **created** for timestamp ts (N is incremented)
 - $WTM_1(x), \dots, WTM_N(x)$ are the new versions, kept sorted from oldest to youngest
 - NB: this version is used in real systems based, e.g., on *snapshot isolation* (see later) and **in the exercises**

Example in practice (for the exam): TS-Multi under Snapshot Isolation

Assume $\text{RTM}(x) = 7$, $N=1$, $\text{WTM}_1(x) = 4$

<i>Request</i>	<i>Response</i>	<i>New Value</i>
$r_6(x)$	ok	
$r_8(x)$	ok	$\text{RTM}(x) = 8$
$r_9(x)$	ok	$\text{RTM}(x) = 9$
$w_8(x)$	no	T_8 killed
$w_{11}(x)$	ok	$\text{WTM}_2(x) = 11$, $N=2$
$r_{10}(x)$	ok on x_1	$\text{RTM}(x) = 10$ (no longer killed)
$r_{12}(x)$	ok on x_2	$\text{RTM}(x) = 12$
$w_{14}(x)$	ok	$\text{WTM}_3(x) = 14$, $N=3$
$w_{13}(x)$	no	but in the exercises, T_{13} is killed

CSR, VSR, 2PL, TSmono, TSmulti



- Versions: X_0 (original value), X_1 , X_2
- r_1 reads X_1 [$WTM_k(x) \leq ts < WTM_{k+1}(x)$]
- Schedule not in VSR
 - T_1, T_2 different reads-from relation for $r_1(x)$
 - T_2, T_1 different final write relation

Snapshot Isolation (SI)

- The realization of multi-TS gives the opportunity to introduce into DBMSs another isolation level,
SNAPSHOT ISOLATION
- In this level, only the timestamp of write operations is used
- Every transaction reads the version consistent with its timestamp (i.e., the version that existed when the transaction started a.k.a. **snapshot**)
- If the scheduler detects that the writes of a transaction conflict with writes of other concurrent transactions after the snapshot timestamp, it aborts
 - It is yet another case of optimistic concurrency control

Anomalies in Snapshot Isolation

- Snapshot isolation does not guarantee serializability
 - T1: update Balls set Color=White where Color=Black
 - T2: update Balls set Color=Black where Color=White
- Serializable executions of T1 and T2 will produce a final configuration with balls that are either all white or all black
- An execution under Snapshot Isolation in which the two transactions start with the same snapshot will just swap the two colors
- This anomaly is called **write skew**

Concurrency control in some commercial DBMS

- ORACLE
 - https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm
- MYSQL
 - <https://dev.mysql.com/doc/refman/8.0/en/locking-issues.html>
- IBM DB2
 - https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.7.0/com.ibm.db2.luw.admin.perf.doc/doc/c0054923.html
- MONGODB
 - <https://docs.mongodb.com/manual/faq/concurrency/>

Databases 2

Triggers

Active Databases

- Outline
 - Introduction: reactive behaviors
 - Trigger definition in SQL:1999
 - Execution semantics and properties of trigger-based systems
 - Examples
 - Evolution of triggers
 - Trigger definition in some commercial DBMSs

Definition and history

- TRIGGERS in DBMS:
 - “A set of actions that are automatically executed when an INSERT, UPDATE, or DELETE operation is performed on a specific table”
 - From “passive” to “active” behaviors
- 1975: Idea of “integrity constraints”
- Mid 1980-1990: research in constraints & triggers
- SQL-92: constraints
 - Key constraints, referential integrity, domain constraints
 - DECLARATIVE SPECIFICATION
- SQL-99: triggers/active rules
 - PROCEDURAL SPECIFICATION
 - Widely-varying support in products, with different execution semantics

The Trigger Concept

- Event-Condition-Action (ECA) paradigm
 - whenever an event E occurs
 - if a condition C is true
 - then an action A is executed
- Triggers are executed in addition to integrity constraints and allow one to check complex conditions
- They are compiled and stored in the DBMS similarly to stored procedures but are executed automatically based on the occurrence of events rather than invoked by the client

Event-Condition-Action

- Event
 - Normally a modification of the database status: insert, delete, update
- Condition
 - A predicate that identifies those situations in which the execution of the trigger's action is required
- Action
 - A generic update statement or a stored procedure
 - Usually database updates (insert – delete – update)
 - Can include also error notifications

Active database – an example

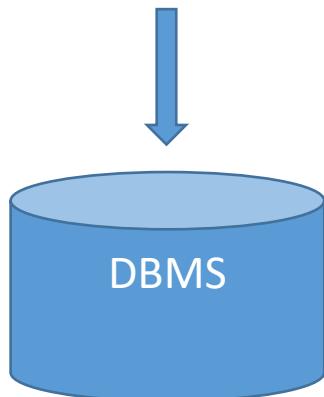
Student (StudID, Name, Email, ...)

Exam (StudID, CourseID, Date, Grade, ...)

Alerts (TS, StudID)

- Log an alert for the student if the Grade is updated

Update of Grade
in Exam

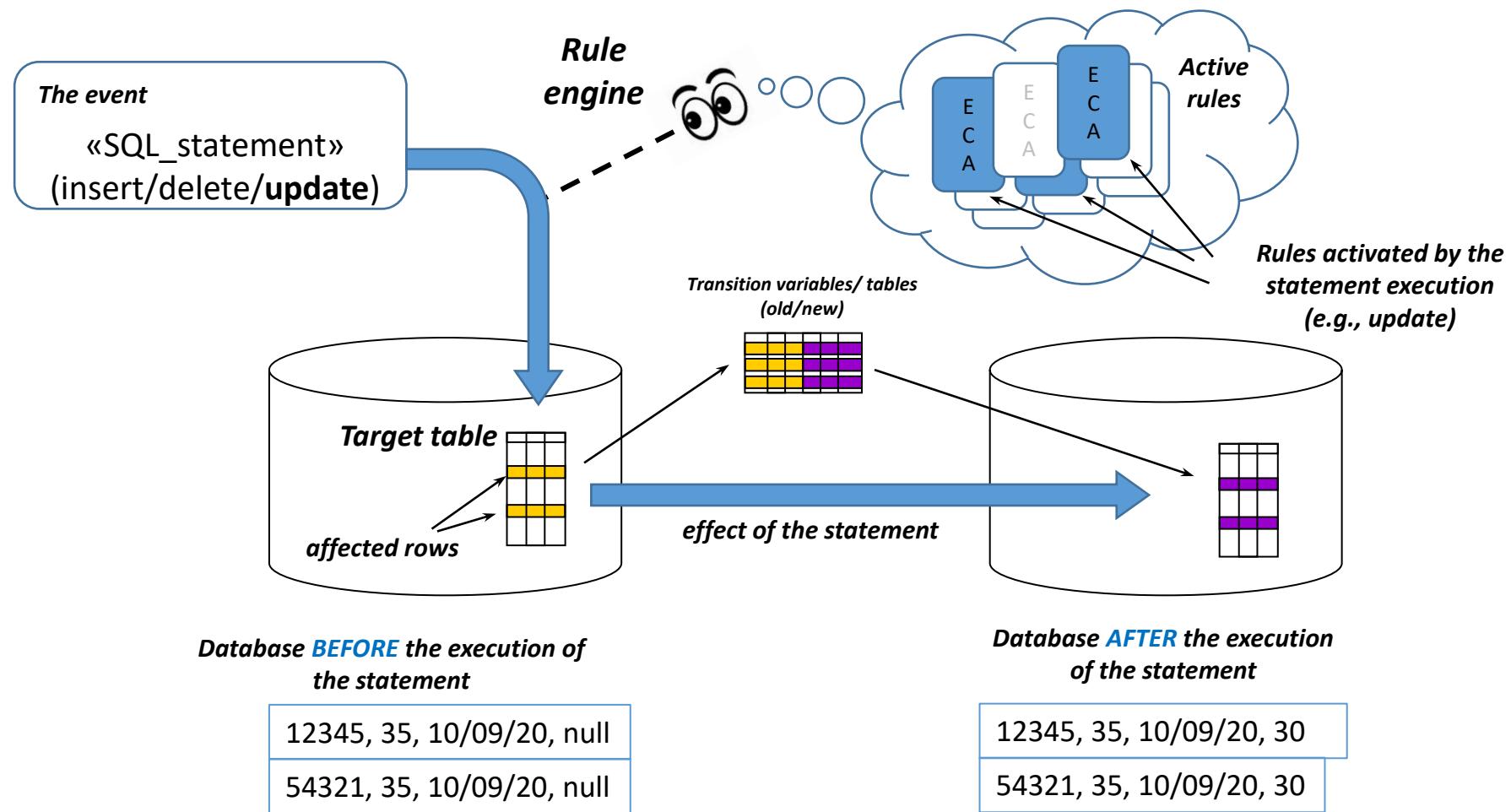


E: Update of Grade on Exam
C: New Grade <> from the previous grade
A: Insert into Alerts

← Insert into Alerts

The big picture

```
UPDATE Exam SET Grade="30" WHERE StudID = "12345" OR  
StudID = "54321" and CourseID="35";
```



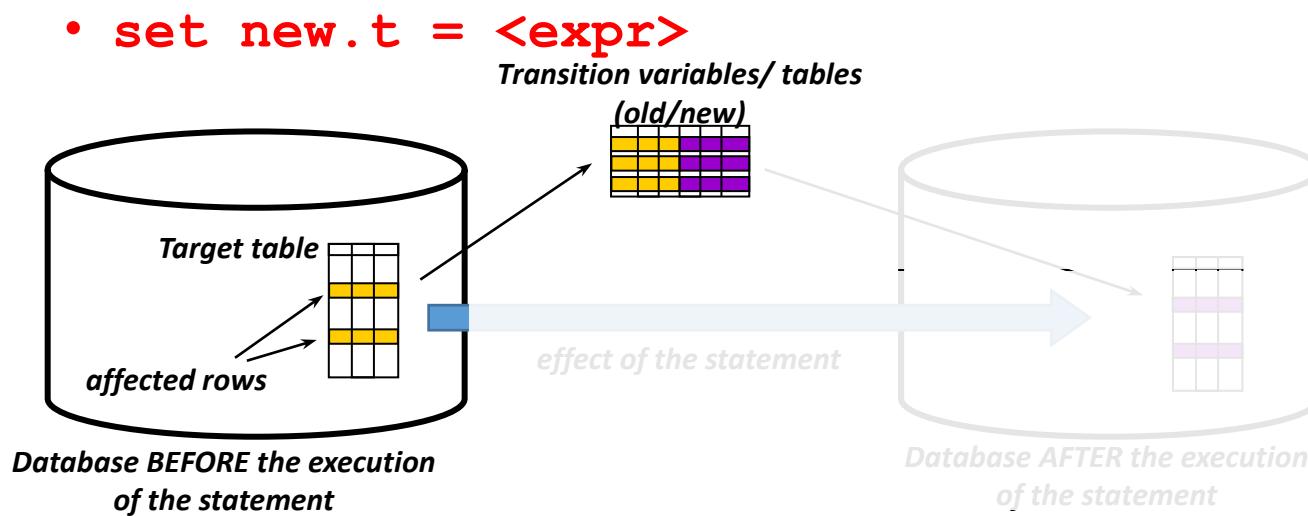
Triggers in SQL:1999, Syntax

```
create trigger <TriggerName>
{ before | after }
E { insert | delete | update [of <Column>] } on <Table>
referencing { [ old table [as] <OldTableAlias> ]
              [ new table [as] <NewTableAlias> ] |
              [ old [row] [as] <OldTupleName> ]
              [ new [row] [as] <NewTupleName> ] }
[ for each { row | statement } ]
C [ when <Condition> ]
A <SQLProceduralStatement>
```

Execution modes: before or after

- **BEFORE**

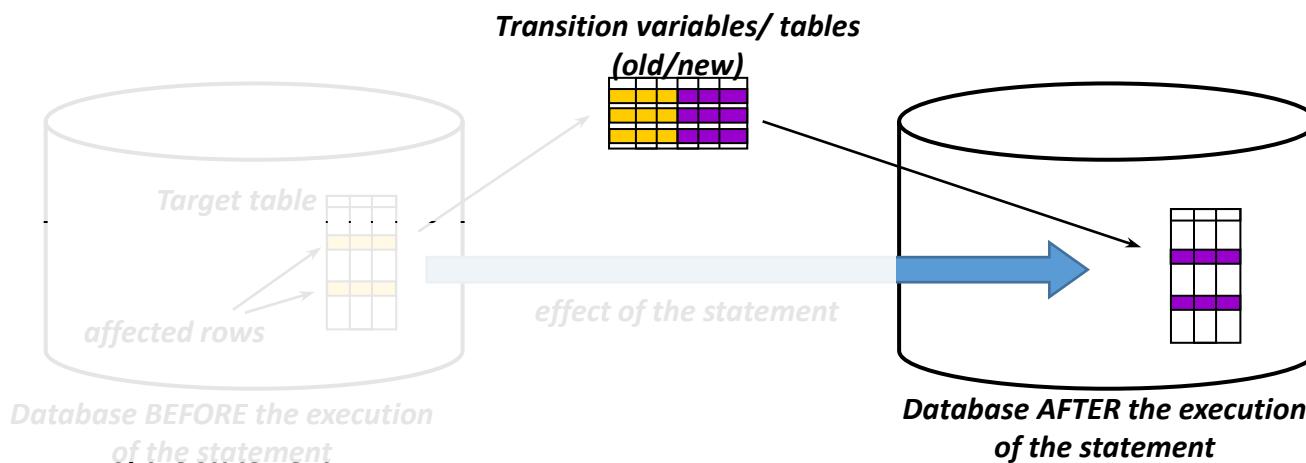
- The action of the trigger is executed **before** the database status change (if the condition holds)
- Used to validate a modification before it takes place and possibly condition its effect
- Safeness constraint: before triggers cannot update the database directly, but can affect the transition variables in row-level granularity
 - **set new.t = <expr>**



Execution modes: before or after

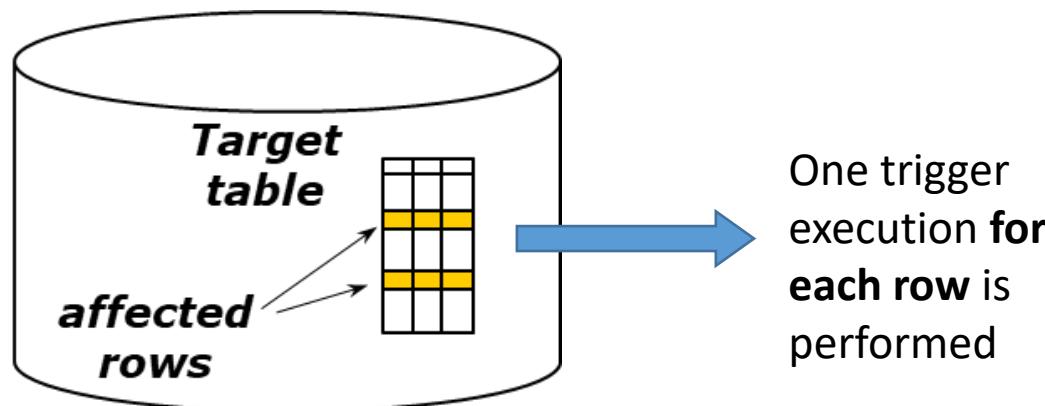
- **AFTER**

- The action of the trigger is executed **after** the modification of the database (if the condition holds)
- It is the most common mode, suitable for most applications



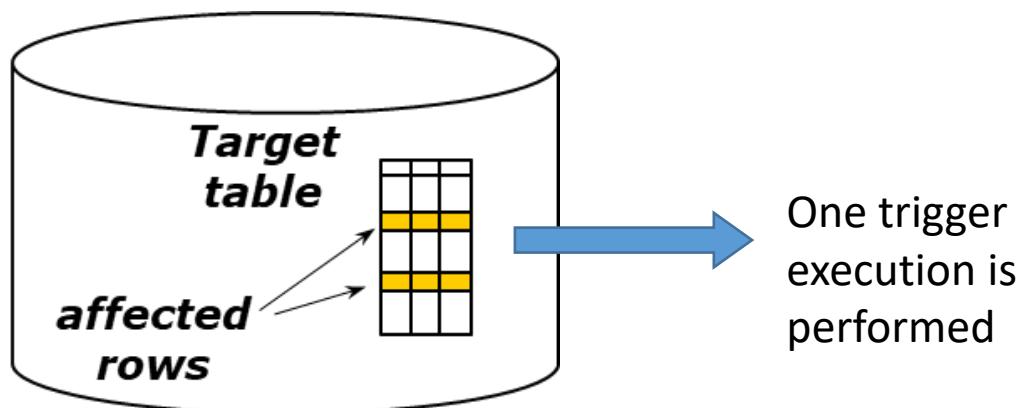
Granularity of events

- **Row-level granularity** (for each row)
 - The trigger is considered and possibly executed **once for each tuple** affected by the activating statement
 - Writing row-level triggers is simpler, but can be less efficient



Granularity of events

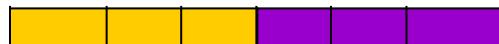
- **Statement-level granularity** (for each statement)
 - The trigger is considered and possibly executed only **once for each activating statement**, independently of the number of affected tuples in the target table (even if no tuple is affected!)
 - Closer to the traditional approach of SQL statements, which are normally set-oriented



Transition variables

- Special variables denoting the before and after state of the modification with a syntax that depends on the granularity
 - **Row-level:** tuple variables **old** and **new** represent the value respectively before and after the modification of the row (i.e., tuple) under consideration
 - **Statement-level:** table variables (**old table** and **new table**) contain respectively the old and the new values of all the affected rows (tuples)
- Variables **old** and **old table** are undefined in triggers whose event is **insert**
- Variables **new** and **new table** are undefined in triggers whose event is **delete**

*Transition variables
(old/new)*



*Transition tables
(old table/new table)*

A simple example: data replication

- Table T2 is a replica of table T1: whenever an update occurs on T1, this is replicated on T2
- Replication is realized by means of triggers

- Table T1

ID	VALUE
1	10
2	15



- Table T2

ID	VALUE
1	10
2	15

Insertion of a new tuple into T1

- Table T1:

ID	VALUE
1	10
2	15
3	20

- Table T2:

ID	VALUE
1	10
2	15
3	20

transition variable **new**

ID	VALUE
3	20

```
CREATE TRIGGER REPLIC_INS
AFTER INSERT ON T1
FOR EACH ROW
INSERT INTO T2 VALUES (new.ID, new.VALUE);
```

Deletion of a tuple from T1

- Table T1:

ID	VALUE
1	10
2	15
3	20

- Table T2:

ID	VALUE
1	10
2	15
3	20

transition variable **old**

ID	VALUE
2	15

```
CREATE TRIGGER REPLIC_DEL
AFTER DELETE ON T1
FOR EACH ROW
DELETE FROM T2 WHERE T2.ID = old.ID;
```

Update of just the VALUE of a tuple in T1 (and ID is unchanged)

- Table T1:
- Table T2:

ID	VALUE
1	10 5
2	15
3	20

transition variables

old

ID	VALUE
1	10

ID	VALUE
1	10 5
2	15
3	20

new

ID	VALUE
1	5

```
CREATE TRIGGER REPLIC_UPD
AFTER UPDATE OF VALUE ON T1
WHEN new.ID = old.ID
FOR EACH ROW
UPDATE T2 SET T2.VALUE = new.VALUE
WHERE T2.ID = new.ID;
```

Handling all cases of UPDATE

- The previous trigger (correctly) won't fire when the ID of a tuple changes
 - However, a robust implementation must consider this case, too
- In order to avoid too much clutter in the code, in these slides we only consider updates involving a single, specific attribute
 - In the real world (and in exams, too), we would need to handle all cases
- Homework: rewrite the previous trigger so that it also includes the (unlikely) case of an ID change

Conditional replication

- Variant: Table T2 is a replication of table T1, **but only the tuples whose value is ≥ 10 are replicated**
- All events affecting the replica must be treated
- Insertion operation:

```
CREATE TRIGGER CON_REPL_INS --new relevant tuple, replicate
AFTER INSERT ON T1
FOR EACH ROW
WHEN (new.VALUE >= 10)
INSERT INTO T2 VALUES (new.ID, new.VALUE);
```

- Deletion:

```
CREATE TRIGGER Cond_REPL_DEL -- propagate deletion
AFTER DELETE ON T1
FOR EACH ROW
WHEN (old.VALUE >= 10) -action execution only for relevant tuples
DELETE FROM T2 WHERE T2.ID = old.ID;
```

Conditional replication

- Modification (only `value` is modified and `id` stays the same)

```
CREATE TRIGGER Cond_REPL_UPD_1 -- new relevant tuple, replicate
AFTER UPDATE OF VALUE ON T1 WHEN new.ID = old.ID
FOR EACH ROW
WHEN (old.VALUE < 10 AND new.VALUE >= 10)
INSERT INTO T2 VALUES (new.ID, new.VALUE);
```

```
CREATE TRIGGER Cond_REPL_UPD_2 -- already replicated tuple changed, propagate
AFTER UPDATE OF VALUE ON T1 WHEN new.ID = old.ID
FOR EACH ROW
WHEN (old.VALUE >= 10 AND new.VALUE >= 10 AND old.VALUE != new.VALUE)
UPDATE T2 SET T2.VALUE = new.VALUE WHERE T2.ID = new.ID
```

```
CREATE TRIGGER Cond_REPL_UPD_3 -- replicated tuple no longer relevant: delete
AFTER UPDATE OF VALUE ON T1 WHEN new.ID = old.ID
FOR EACH ROW
WHEN (old.VALUE >= 10 AND new.VALUE < 10)
DELETE FROM T2 WHERE T2.ID = new.ID;
```

BEFORE trigger

- Prevent values to be updated to negative values.
In such a case, they are set to 0
- Statement:

```
UPDATE T1 SET T1.VALUE = -8 WHERE T1.ID = 5
```

- Trigger

```
CREATE TRIGGER NO_NEGATIVE_VALUES  
BEFORE UPDATE of VALUE ON T1  
FOR EACH ROW  
WHEN (new.VALUE < 0)  
SET new.VALUE=0; -- this "modifies the modification"
```

old	
ID	VALUE
5	6

new	
ID	VALUE
5	-8

BEFORE vs AFTER

```
CREATE TRIGGER NO_NEGATIVE_VALUES
AFTER UPDATE of VALUE ON T1
FOR EACH ROW
WHEN (new.VALUE < 0)
UPDATE T1 SET VALUE = 0 WHERE ID=new.ID;
```

Not allowed in some DBMSs



- Apparently an AFTER trigger does the same thing as the BEFORE trigger
- **But it is not equivalent:** in the example, with the BEFORE trigger there is only 1 UPDATE statement, with the AFTER trigger there are 2 UPDATE statements
- BEFORE triggers are more efficient if the goal is to "modify a modification"
- Some DBMSs (e.g., MySQL) disallow a trigger to update the table affected by the triggering event!

row vs. statement: DELETE event

- Conditional replication with statement level triggers
- Statement:

```
DELETE FROM T1 WHERE VALUE >= 5;
```

ID	VALUE
1	3
2	5
3	20

- Trigger:

```
CREATE TRIGGER ST_REPL_DEL  
AFTER DELETE ON T1  
REFERENCING OLD TABLE AS OLD_T  
FOR EACH STATEMENT --all tuples considered at once
```

old table

ID	VALUE
2	5
3	20

```
DELETE FROM T2 WHERE T2.ID IN  
(SELECT ID FROM OLD_T); -- no need to add where OLD_T.value >=10
```

row vs. statement: INSERT event

- Statement:

```
INSERT INTO T1 (Id, Value) VALUES (4, 5), (5, 10), (6, 20);
```

- Trigger:

```
CREATE TRIGGER ST_REPL_INS
```

```
AFTER INSERT ON T1
```

```
REFERENCING NEW TABLE AS NEW_T
```

FOR EACH STATEMENT --all tuples considered at once

```
INSERT INTO T2
```

```
(SELECT ID, VALUE
```

```
FROM NEW_T WHERE NEW_T.VALUE >= 10);
```

new table

ID	VALUE
4	5
5	10
6	20

row vs. statement: UPDATE event

- Statement 1: double the value

```
UPDATE T1 SET value = 2 * value;
```

Initial DB state

T1 / before stm 1/2

ID	VALUE
1	3
2	5
3	18
4	40

(initial) T2

ID	VALUE
3	18
4	40

new_T / after stm 1

ID	VALUE
1	6
2	10
3	36
4	80

T2

ID	VALUE
2	10
3	36
4	80

- Statement 2: half the value

```
UPDATE T1 SET value = 0.5 * value;
```

new_T / after stm 2

ID	VALUE
1	1.5
2	2.5
3	9
4	20

T2

ID	VALUE
4	20

row vs. statement: UPDATE event

```
CREATE TRIGGER REPLIC_INS
```

```
AFTER UPDATE ON T1
```

```
REFERENCING OLD TABLE AS OLD_T NEW TABLE AS NEW_T
```

FOR EACH STATEMENT

```
DELETE FROM T2 --delete all updated rows
```

```
WHERE T2.ID IN (SELECT ID FROM OLD_T);
```

```
INSERT INTO T2 --reinsert only relevant rows
```

```
(SELECT ID, VALUE FROM NEW_T
```

```
WHERE NEW_T.VALUE >= 10);
```

ID	VALUE
1	3
2	5
3	18
4	40

ID	VALUE
3	18
4	40

New_T / after stm 1

ID	VALUE
1	6
2	10
3	36
4	80

ID	VALUE
2	10
3	36
4	80

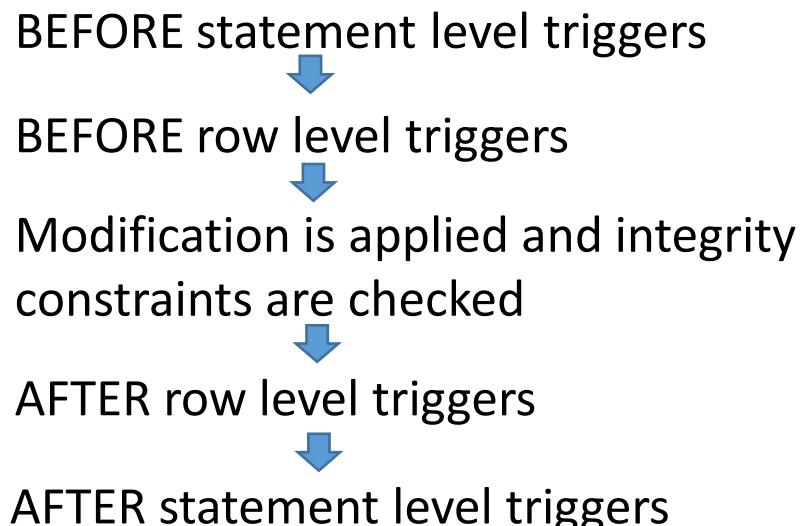
New T / after stm 2

ID	VALUE
1	1.5
2	2.5
3	9
4	20

ID	VALUE
4	20

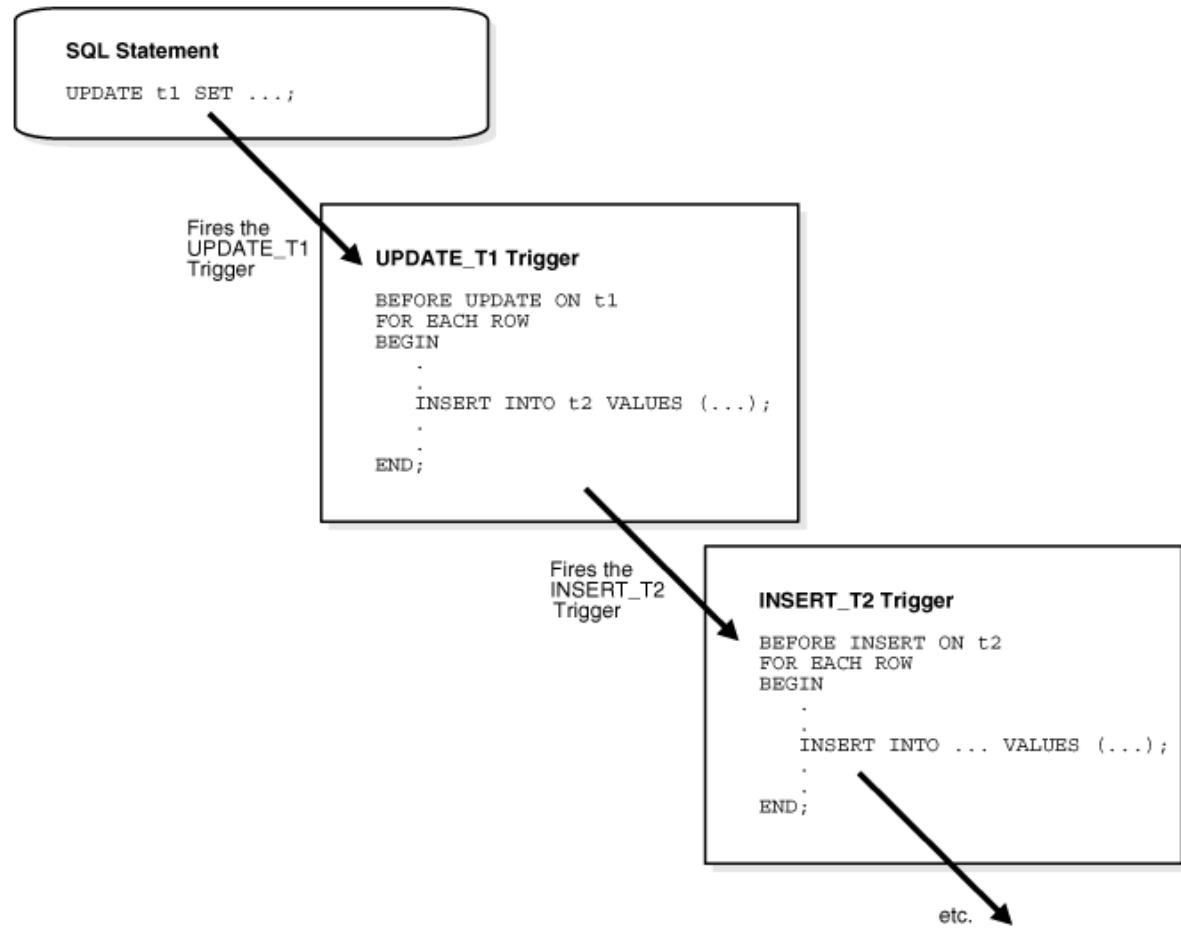
Multiple triggers on same event

- If several triggers are associated with the same event, SQL:1999 prescribes the execution sequence
- If there are several triggers in the same category, the order of execution depends on the system implementation
 - e.g., based on the definition time (older triggers have higher priority) or based on the alphabetical order of the name



Cascading and recursive cascading

- The action of a trigger may cause another trigger to fire
- **Cascading**: when the action of T1 triggers T2 (also called nesting)
- **Recursive cascading**: a statement S on table T starts a cascading of triggers that generates the same event S on T (also called looping)



Termination analysis

- It is important to ensure that recursive cascading does not produce undesired effects
- **Termination**: for any initial state and any sequence of modifications, a final state is always produced (infinite activation cycles are not possible)
- The simplest check exploits the **triggering graph**
 - A node i for each trigger T_i
 - An arc from a node i to a node j if the execution of trigger T_i 's action may activate trigger T_j
- The graph is built with a simple syntactic analysis
- If acyclic, the system is guaranteed to terminate
- If cycles exist, triggers may terminate or not
 - Acyclicity is **sufficient** for termination, not necessary

Example

Employee (RegNum, Name, Salary, Contribution, DeptN, ...)

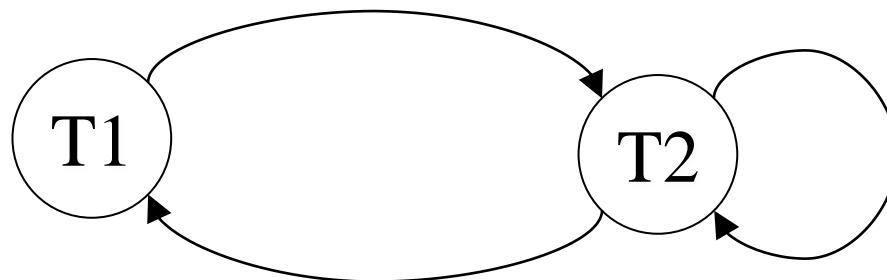
- T1:

```
create trigger AdjustContributions
after update of Salary on Employee
referencing new table as NewEmp
for each statement
update Employee
set Contribution = Salary * 0.8
where RegNum in (select RegNum from NewEmp)
```

- T2:

```
create trigger CheckOverallBudgetThreshold
after update on Employee - salary OR contribution updated
for each statement
when (select sum(Salary+Contribution) from Employee) > 50000
update Employee
set Salary = 0.9 * Salary;
```

Terminating cyclic triggering graph



- There are two cycles, but the system is terminating
- What if you invert the comparison in T2's condition?
 - **when (select sum(Salary+Contribution)
from Employee) < 50000**

How real systems work

- MySQL
 - A stored function or trigger cannot modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger
 - <https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html>
- Postgres
 - If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.
 - <https://www.postgresql.org/docs/9.1/trigger-definition.html>
- MS SQL Server
 - Triggers are nested when a trigger performs an action that initiates another trigger. These actions can initiate other triggers, and so on. Triggers can be nested up to 32 levels. An AFTER trigger does not call itself recursively unless the RECURSIVE_TRIGGERS database option is set.
 - <https://docs.microsoft.com/en-us/sql/relational-databases/triggers/create-nested-triggers?view=sql-server-ver15>
- Oracle
 - When a statement in a trigger body causes another trigger to be fired, the triggers are said to be cascading. Oracle Database allows up to 32 triggers to cascade at any one time. However, you can effectively limit the number of trigger cascades using the initialization parameter OPEN_CURSORS, because a cursor must be opened for every execution of a trigger.
 - https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_tr.htm
- IBM Informix
 - Complicated...
 - <https://www.ibm.com/docs/en/informix-servers/12.10?topic=statement-cascading-triggers>

Trigger applications

- Triggers add powerful data management capabilities in a transparent and reusable manner
 - Databases can be enriched with “business and management rules” that would otherwise be distributed over all applications
- However, understanding the interactions between triggers is rather complex
- Some DBMS vendors use triggers to implement internal services
- Examples:
 - Data replication
 - Integrity constraint management not supported by declarative SQL primitives (e.g., constraints on data changes)
 - Materialized view maintenance

Example: book sales

- Consider the following relational schema:

BOOK (Isbn, Title, BSoldCopies)

WRITING (Isbn, Name)

AUTHOR (Name, **ASoldCopies**)

- Define a set of triggers for keeping ASoldCopies in

AUTHOR updated with respect to:

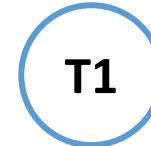
- updates on BSoldCopies in BOOK
- insertion of new tuples in the WRITING relation

Book sales: solution outline

`BOOK(Isbn, Title, BSoldCopies)`

`WRITING(Isbn, Name)`

`AUTHOR(Name, ASoldCopies)`



- T1
 - Event: update of BSoldCopies on BOOK
 - Condition: none
 - Action: update ASoldCopies of AUTHOR for existing authors
- T2
 - Event: insert on WRITING
 - Condition: none
 - Action: update ASoldCopies of AUTHOR for new book's author
- T3 ?
 - Event: insert on BOOK (or on AUTHOR)
 - Not necessary: to insert in WRITING a tuple of BOOK (and of AUTHOR) must already exist (due to referential integrity)

No cascading

Book update event

Assume for simplicity that `Isbn` cannot be updated

```
create trigger UpdateSalesAfterNewSale  
after update of BSoldCopies on Book  
for each row  
update Author  
set ASoldCopies = ASoldCopies  
    + new.BSoldCopies - old.BSoldCopies  
where Name in (select Name  
                from Writing  
                where Isbn = new.Isbn)
```

Amount of
new sales

WARNING!! `Author.ASoldCopies` is NOT equal to `Book.BSoldCopies`.

Writing insert event

```
create trigger UpdateSalesAfterNewAuthorship
after insert on Writing
for each row
update Author
set ASoldCopies = ASoldCopies +
(select BSoldCopies from Book where Isbn = new.Isbn)
where Name = new.Name
```

Example: materialized views

- A view is a virtual table defined with a query stored in the database catalog and then used in queries as if it were a normal table
- A simple example: overall personnel cost of each department:

```
CREATE VIEW deptcost AS  
SELECT d.DeptNum as dept, coalesce(sum(e.salary), 0) as totCost  
FROM dept d LEFT JOIN emp e ON e.dept = d.DeptNum  
GROUP BY d.DeptNum;
```

- The view shows the personnel cost of every department

dept	
deptNum	Name
1	DEIB
2	DICA

emp		
RegNum	Salary	Dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2

deptcost	
dept	totCost
1	264
2	168

View materialization

- When a view is mentioned in a SELECT query the query processor rewrites the query using the view definition, so that the actually executed query only uses the **base tables** of the view
- When the queries to a view are more frequent than the updates on the base tables that change the view content, then **view materialization** can be an option
 - Storing the results of the query that defines a the view in a table
- Some systems support the CREATE MATERIALIZED VIEW command, which makes the view automatically materialized by the DBMS
- An alternative is to implement the materialization by means of triggers

Incremental view maintenance

- Not all modifications of Employee affect the view, and most modifications only affect a small part of the materialized data:
 - **Update of Idemp:** no effect
 - **Insertion of an employee, deletion of an Employee, update of one salary attribute of employee:** affects only one tuple in deptcost
 - **Update of dept attribute of one employee:** affects only two tuples in deptcost
 - **Insertion and deletion of a department:** affects only one tuple in deptcost
- When the effort to deal with the variation in the view is smaller than that of recomputing it from scratch, an incremental approach is preferable

dept	
deptNum	Name
1	DEIB
2	DICA

emp		
Idemp	salary	dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2

deptcost	
dept	totCost
1	264
2	168

View maintenance: solution sketch

- T1
 - Event: insert on EMP
 - Condition: none
 - Action: increase totCost of deptCost
- T2
 - Event: delete on EMP
 - Condition: none
 - Action: decrease totCost of deptCost
- T3
 - Event: update of **salary** on EMP
 - Condition: dept is unchanged
 - Action: update totCost of deptCost
- T4
 - Event: update of **dept** of EMP
 - Condition: none
 - Action: increase totCost of new dept in deptCost, decrease totCost of old dept in deptCost
- T5
 - Event: insert on DEPT
 - Condition: none
 - Action: insert tuple and set totCost to 0 in DeptCost
- T6
 - Event: delete on DEPT
 - Condition: none
 - Action: delete on deptCost (can be managed with referential integrity)

No cascading: events on base tables, actions on materialized view table

Insertion /deletion of employees

```
create trigger Incremental_InsEmp  
after insert on emp  
for each row  
update deptcost  
    set totCost = totCost + new.salary  
    where dept = new.dept;  
  
create trigger Incremental_DelEmp  
after delete on emp  
for each row  
update deptcost  
    set totCost= totCost - old.salary  
    where dept = old.dept;
```

row-level triggers

If more than one employee is inserted(deleted) by the same SQL command, each affected tuple impacts only one department in the materialized view

Salary update (dept unchanged)

```
create trigger Incremental_SalaryUpdate  
after update of salary on emp  
when old.dept = new.dept - dept of employee unchanged  
for each row  
update deptcost  
set totCost = totCost - old.salary + new.salary  
where dept = new.dept;
```

idemp	salary	dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2
6	80	2

dept	totCost
1	264
2	243

248

Applies the “salary variation”...

...only to the department to which the updated employee is affiliated (using **old.dept** would be equally correct, as this is not changed)

Update of dept (moving the employee)

```
create trigger Incremental_DeptChange
after update of dept on emp
for each row
begin
    update deptcost
        set totCost = totCost + new.salary
        where dept = new.dept;
    update deptcost
        set totCost = totCost - old.salary
        where dept = old.dept;
end;
```

*The sum is incremented
(resp. decremented) in
the new (resp. old)
department*

idemp	salary	dept
1	100	1
2	80	2
3	74	1
4	90	1
5	88	2
6	80	2

1

dept	totCost
1	264
2	168

Creation/deletion of a department

```
create trigger Incremental_deptinsert
after insert on dept
for each row
begin
    insert into deptcost values(new.deptNum, 0);
end;

--can be managed with FK constraint
create trigger Incremental_deptdelete
after delete on dept
for each row
begin
    delete from deptcost where dept = old.deptNum;
end;
```

Triggers and integrity

- Integrity maintenance triggers (as all triggers) interact with referential integrity
- The previous incremental triggers assume referential integrity constraints between dept and emp

```
CREATE TABLE emp (
    idemp int NOT NULL AUTO_INCREMENT,
    salary int DEFAULT NULL,
    dept int NOT NULL,
    PRIMARY KEY (idemp),
```

```
CONSTRAINT 'empdept' FOREIGN KEY (dept)
REFERENCES dept (DeptNum) ON DELETE CASCADE)
```

- Deleting a department cascades the deletion on the affiliated employees
- Assigning an employee to a non existing department produces a constraint violation

Example: sports competition

- Given the database

ATHLETE (ATHL_ID, **personal_record**, **qualified**)

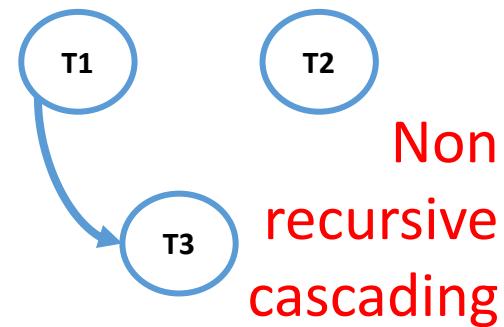
REGIONAL_COMPETITION (COMP_ID, **number_of_enrolled**, **winner**)

RESULT (COMP_ID, ATHL_ID, **time**)

- At the end of a competition, an application inserts all the results of the athletes. Some triggers react to the insertion of the results of a competition, by:
 - Setting the winner of the competition
 - Updating the personal record if some athlete has improved her/his best time
 - Updating to true the **qualified** attribute of an athlete if s/he has won at least 10 regional competitions

Sports competition: solution outline

- T1
 - Event: insert on RESULT
 - Condition: all results have been inserted into the database
 - Action: update REGIONAL_COMPETITION set winner
- T2
 - Event: insert on RESULT
 - Condition: time < personal record
 - Action: update ATHLETE set personal record
- T3
 - Event: update of winner on REGIONAL_COMPETITION
 - Condition: 10 competitions won
 - Action: update ATHLETE set qualified to true



Trigger T1

```
ATHLETE (ATHL_ID, personal_record, qualified)
REGIONAL_COMPETITION (COMP_ID, number_of_enrolled, winner)
RESULT (COMP_ID, ATHL_ID, time)
```

- Event: insert on RESULT
- Condition: this is the last result & time < all other times
- Action: update REGIONAL_COMPETITION set winner

```
create trigger UpdateWinner
after insert on result
for each row
-- fire only when all results have been inserted
WHEN (SELECT number_of_enrolled FROM regional_competition
      WHERE COMP_ID = new.COMP_ID) =
      (SELECT count(*) FROM result WHERE COMP_ID = new.COMP_ID)
BEGIN
-- the competition winner is the one with the minimum time
  UPDATE regional_competition
    SET winner = (SELECT athl_id from result
                  where comp_id = new.comp_id and
                  time = (SELECT min(time) from result
                          where comp_id = new.comp_id))
    WHERE COMP_ID = new.COMP_ID
END;
```

Trigger T2

```
ATHLETE (ATHL_ID, personal_record, qualified)
REGIONAL_COMPETITION (COMP_ID, number_of_enrolled, winner)
RESULT (COMP_ID, ATHL_ID, time)
```

- Event: insert on RESULT
- Condition: time < personal record or no personal record
- Action: update ATHLETE set personal record

```
create trigger UpdatePersonalRecord
after insert on result
for each row
WHEN
    (SELECT personal_record FROM athlete WHERE
        ATHL_ID=new.ATHL_ID) IS NULL OR
    new.time < (SELECT personal_record FROM athlete WHERE
        ATHL_ID=new.ATHL_ID)
BEGIN
    UPDATE athlete
    SET personal_record=new.time
    WHERE ATHL_ID=new.ATHL_ID
END
```

Trigger T3

```
ATHLETE (ATHL_ID, personal_record, qualified)
REGIONAL_COMPETITION (COMP_ID, number_of_enrolled, winner)
RESULT (COMP_ID, ATHL_ID, time)
```

- Event: update of winner on REGIONAL_COMPETITION
- Condition: 10 competitions won
- Action: update ATHLETE set qualified to true

```
create trigger updateQualification
after update of winner on regional_competition
for each row
WHEN 10 = (SELECT count(*) FROM competition
            where winner = New.winner)
BEGIN
    UPDATE athlete
    SET qualified = true
    WHERE ATHL_ID = NEW.winner
END
```

T1 simplified

```
ATHLETE (ATHL_ID, personal_record, qualified)
REGIONAL_COMPETITION (COMP_ID, number_of_enrolled, winner)
RESULT (COMP_ID, ATHL_ID, time)
```

- Does this simpler version work?

```
create trigger UpdateWinner2
after insert on result
for each row
WHEN (SELECT min(time) FROM result
      WHERE COMP_ID = new.COMP_ID) = new.time AND
      (SELECT number_of_enrolled FROM competition
       WHERE COMP_ID = new.COMP_ID) =
      (SELECT count(*) FROM result
       WHERE COMP_ID = new.COMP_ID)
BEGIN
    UPDATE regional_competition
    SET winner = new.ATHL_ID
    WHERE COMP_ID = new.COMP_ID;
END
```

Inserting rows with multiple statements

- Multiple 1-row inserts

```
insert into result values (1,1,30);
insert into result values (1,2,20); -- this is the winner
insert into result values (1,3,40);
insert into result values (1,4,50);
```

- The trigger is fired and the when condition tested for each row insertion
- When the when condition is tested on the second result, only the first and second results have been inserted
 - Count (*) does not coincide with number_of_enrolled
 - The real winner is missed

Inserting multiple rows in one statement

Postgres

```
insert into result values  
(1,1,30),  
(1,2,20),  
(1,3,40),  
(1,4,50);
```

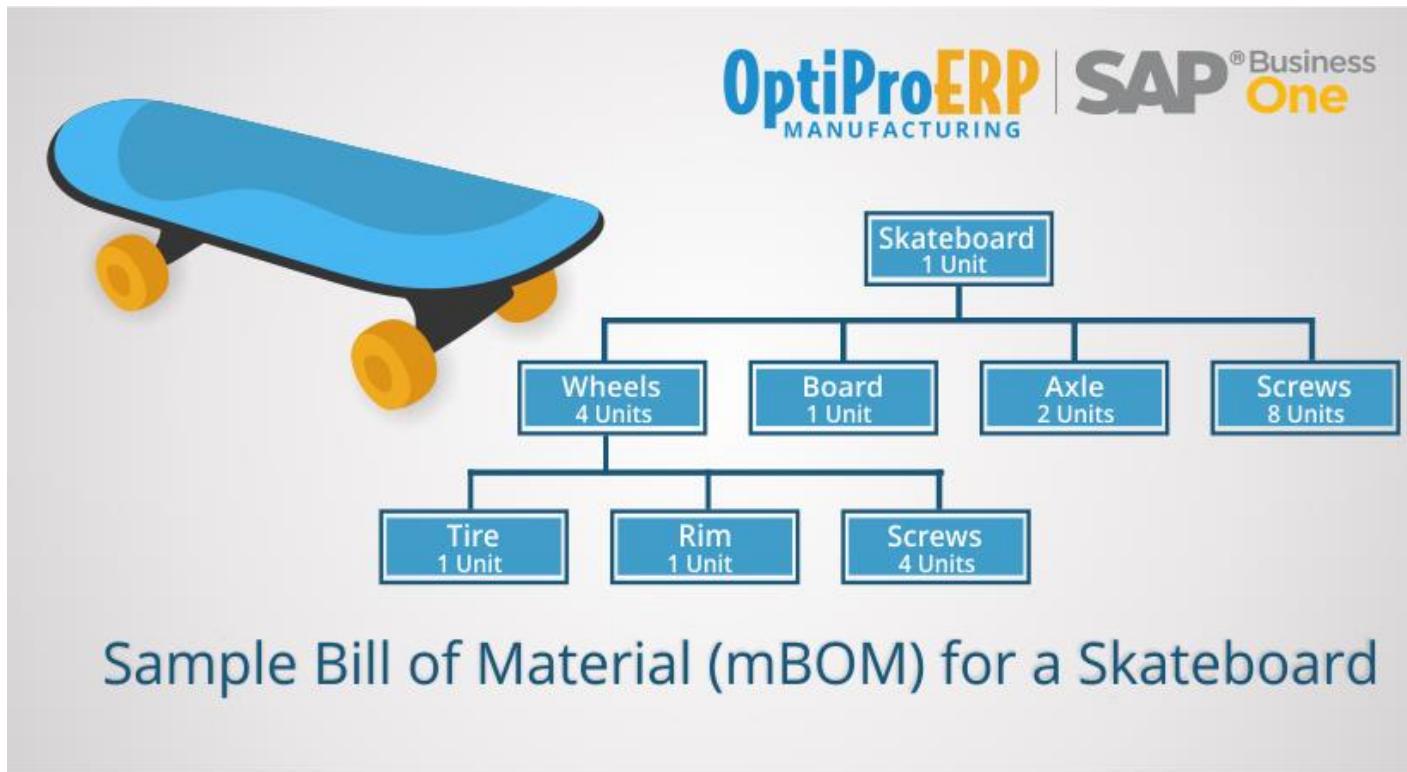
- The trigger is fired for each row
- When the trigger fires on the second row all rows have been inserted
- Count (*) coincides with number_of_enrolled
- Min(time) identifies the winner

MySQL

```
insert into result values  
(1,1,30),  
(1,2,20),  
(1,3,40),  
(1,4,50);
```

- The trigger is fired for each row
- When the trigger fires on the second row only the first and second rows have been inserted
- Count (*) does not coincide with number_of_enrolled
- The real winner is missed

Example: bill of materials (BOM)



Source: <https://www.optiproerp.com/blog/10-types-boms-explained/>

Hierarchy management

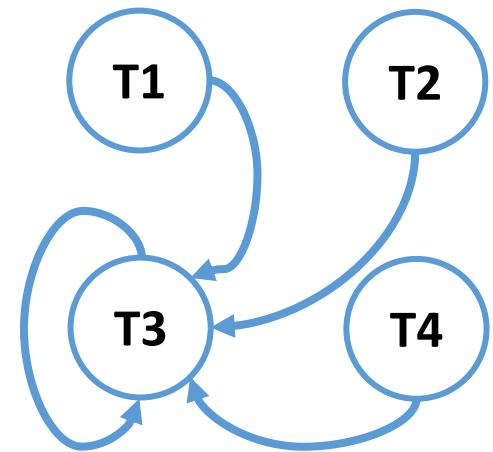
- Example: BOM, a hierarchy of products

Product (ID, Name, SuperProduct, OwnWeight, **TotalWeight)**

- Each product is characterized by a super-product and by an own weight
- The total weight of a product is the sum of the weights of its sub-products (if they exist) and of its own weight
- Root products have:
 - SuperProduct = NULL
- Assume that users can insert and delete products and modify the superproduct of a product
- The total weight can be computed by a recursive view (with recursive and hierarchical queries in SQL:1999)
- The use of triggers to build and maintain the value of the total weight in the hierarchy is good design **ONLY when queries are much more frequent than updates**

Hierarchy management: solution outline

- T1
 - Event: insertion of a new product (user event)
 - Condition: none
 - Action: set total weight of product
- T2
 - Event: deletion of a product (user event)
 - Condition: superProduct not null
 - Action: decrease total weight of super product
- T3
 - Event: **update of total weight** of existing product (trigger event)
 - Condition: superProduct not null
 - Action: **update total weight** of super product
- T4
 - Event: update of superProduct of a product (user event)
 - Condition: superProduct different from previous super product
 - Action: decrease total weight of previous super product, increase total weight of new super product (if not null)
- Deletion of sub-products of a deleted product is managed by referential integrity ON DELETE CASCADE



**Recursive
cascading**

Trigger T1

- Event: insertion of a new product
- Condition: none
- Action: set total weight of product

```
CREATE TRIGGER product_AFTER_INSERT
AFTER INSERT ON product
FOR EACH ROW
BEGIN
UPDATE product SET totalweight = ownweight
WHERE ID = new.id;
END
```

Trigger T2

- Event: deletion of a product
- Condition: superProduct not null
- Action: decrease total weight of super product

```
CREATE TRIGGER product_AFTER_DELETE
AFTER DELETE ON product
FOR EACH ROW
BEGIN
WHEN old.superproduct is not null
UPDATE product SET totalweight = totalweight - old.totalweight
WHERE ID = old.superproduct;
END
```

Trigger T3

- Event: update of total weight of existing product
- Condition: superProduct not null
- Action: update total weight of super product

```
CREATE TRIGGER product_AFTER_UPDATE_TOTALWEIGHT
AFTER UPDATE of totalweight ON product
FOR EACH ROW
BEGIN
WHEN new.superproduct is not null AND new.totalweight != old.totalweight
UPDATE product SET totalweight = totalweight
                    + (new.totalweight - old.totalweight)
WHERE ID = new.superproduct;
END
```

Trigger T4

- Event: update of superProduct of a product
- Condition: superProduct different from previous super product
- Action: decrease total weight of previous super product, increase total weight of new super product (if it exists)

```
CREATE TRIGGER product_AFTER_UPDATE_SUPER
AFTER UPDATE of superproduct ON product
FOR EACH ROW
BEGIN
WHEN
(new.superproduct is null AND) AND old.superproduct is not null ) or
(old.superproduct is null AND new.superproduct is not null) or
(new.superproduct != old.superproduct)
UPDATE product SET totalweight = totalweight - old.totalweight
WHERE ID = old.superproduct;
UPDATE product SET totalweight = totalweight + new.totalweight
WHERE ID = new.superproduct;
END
```

Cascading triggers: a word of caution

- When multiple triggers are activated by the same event, the DBMS establishes a precedence criterion to decide which one to execute first
 - Different vendors implement different policies
- Typically, the delayed triggers will be executed at a later moment
 - To fully understand what happens when those triggers are executed is far from trivial...

Example of what happens on Postgres

- Consider a simple table (`ttest`) with just one row with one attribute (`x`)
 - T1 halves `x` when $x \geq 10$ and T2 increases `x` by 40% when $x \geq 6$

```
CREATE TRIGGER T1 AFTER UPDATE OF x ON ttest
FOR EACH ROW WHEN (NEW.x >= 10)
UPDATE ttest SET X = new.x / 2.0;
```

```
CREATE TRIGGER T2 AFTER UPDATE OF x ON ttest
FOR EACH ROW WHEN (NEW.x >= 6)
UPDATE ttest SET X = new.x * 1.4;
```

```
UPDATE ttest SET x = 12;
```

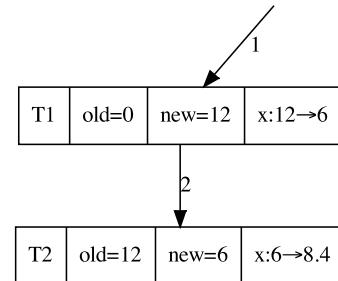
- Let us update x to the value 12 to start the dance...
- Assume its initial value was, e.g., 0

```
UPDATE ttest SET x = 12;
```

			1
T1	old=0	new=12	x:12→6

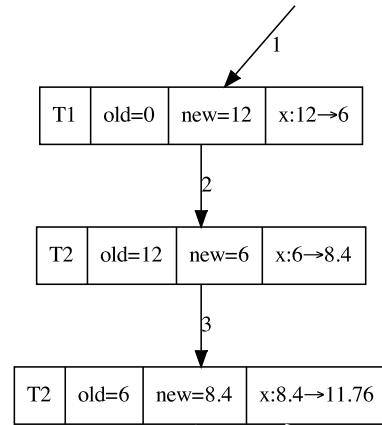
- The update activates both triggers, but T1 takes precedence because it precedes T2 alphabetically.
- The execution of T2 is delayed...
- The initial value of x was 0 (old), the update changed it to 12 (new) and T1 halves its value

UPDATE ttest SET x = 12;



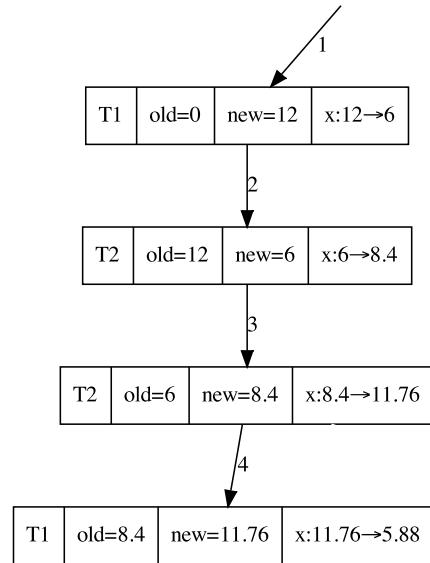
- The change made by T1 activates another instance of T2, which increases x by 40%, from 6 to 8.4

UPDATE ttest SET x = 12;



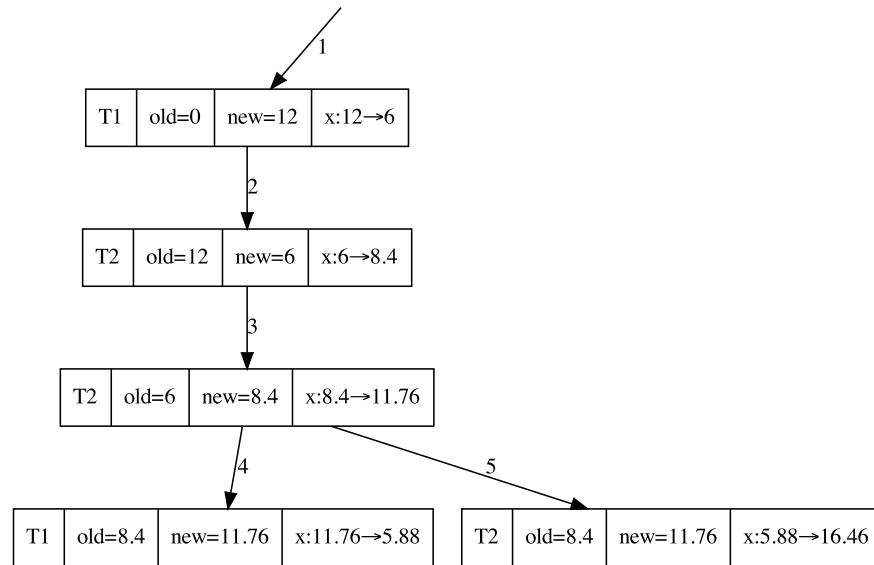
- Then another instance of T2 is executed, changing x from 8.4 to 11.76

UPDATE ttest SET x = 12;



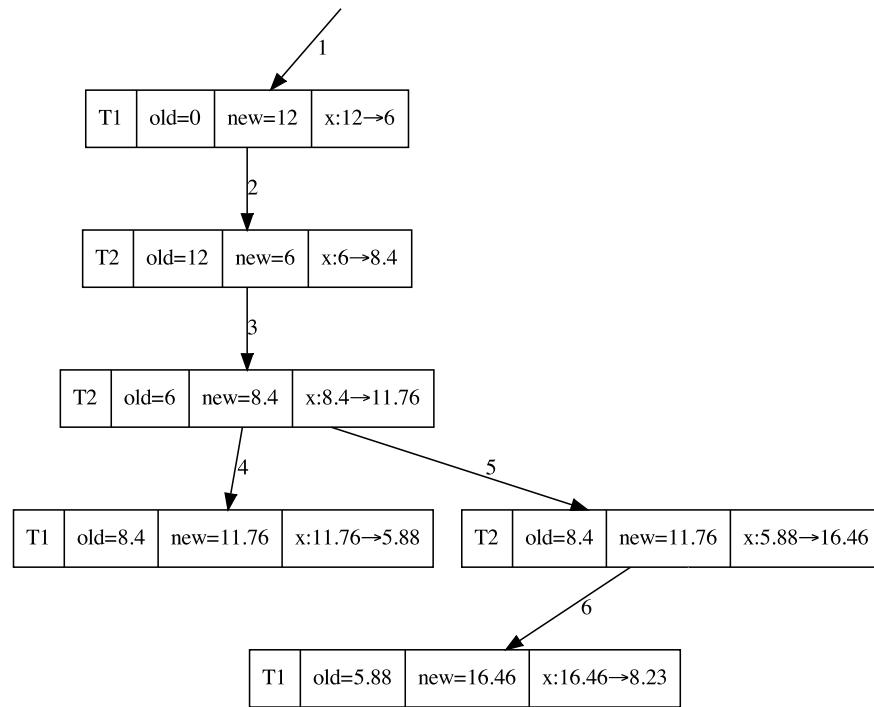
- Now both T1 and T2 are activated, but, again, T1 takes precedence, thereby halving the value of x, and T2 is delayed
- Are we done yet? Not quite...
- Now we can execute the (last) delayed instance of T2

UPDATE ttest SET x = 12;



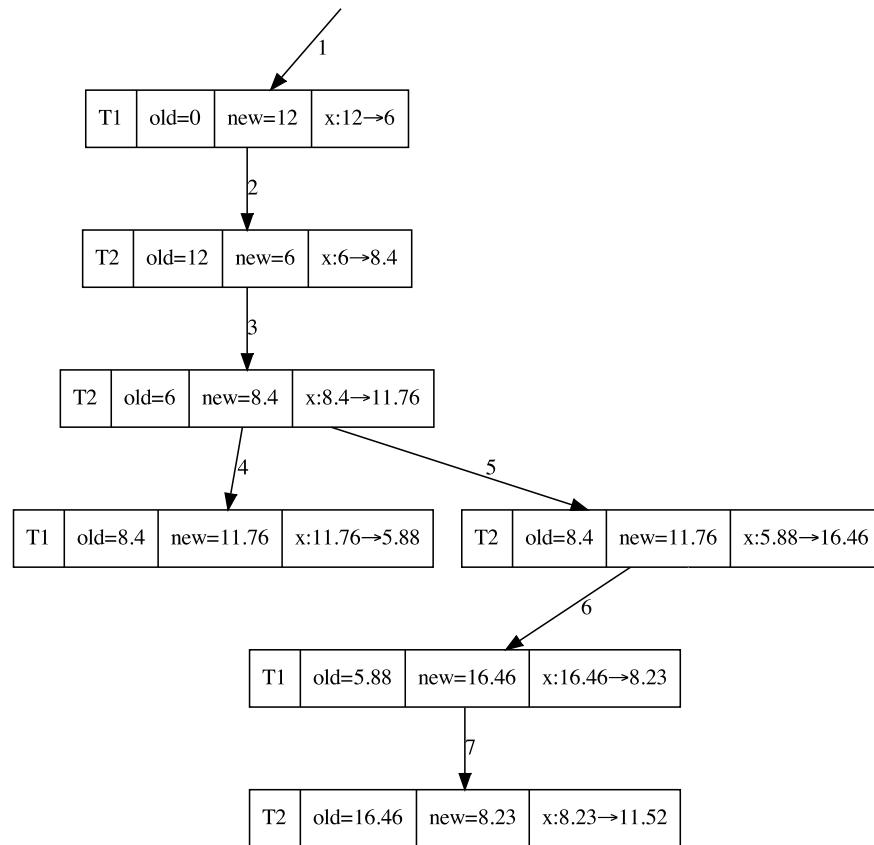
- At step 5, the current value of x is 5.88
- However, the “new” value of x that this instance of T2 sees is still 11.76
 - Indeed, it is executed after step 4, but was activated after step 3
- So, this execution changes x from 5.88 to 16.46
 - Not so obvious!

UPDATE ttest SET x = 12;



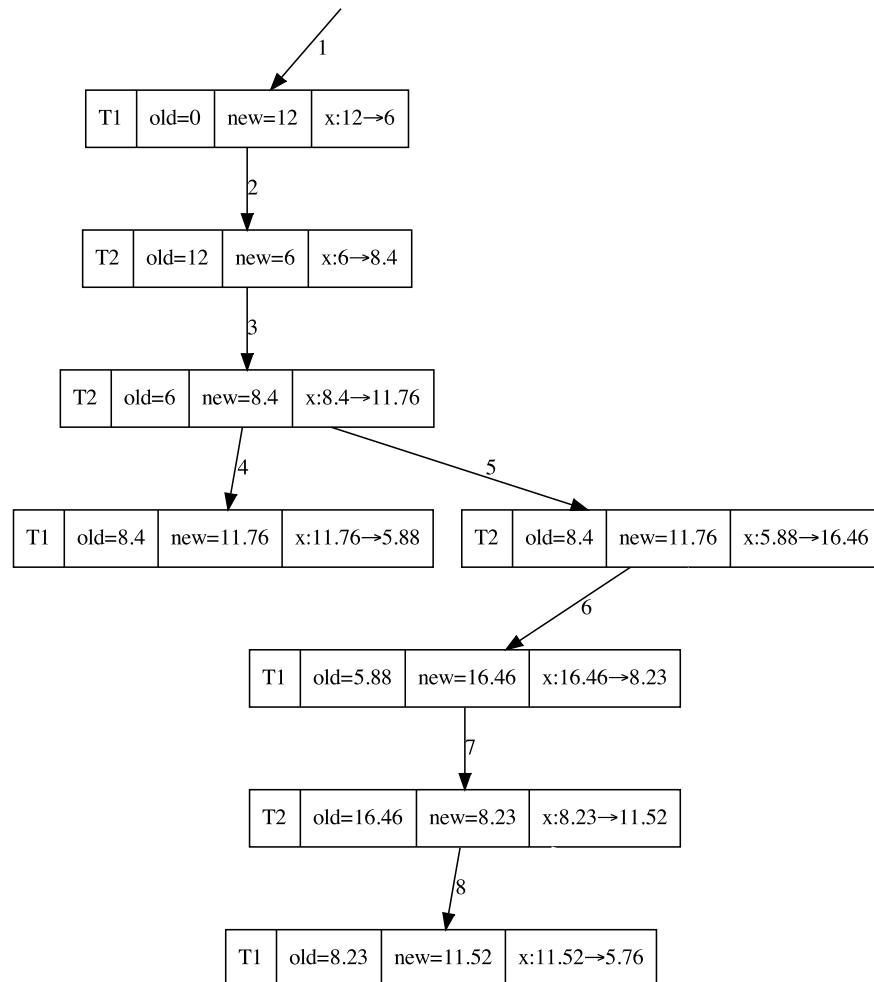
- At this point this chain of activations goes on

UPDATE ttest SET x = 12;



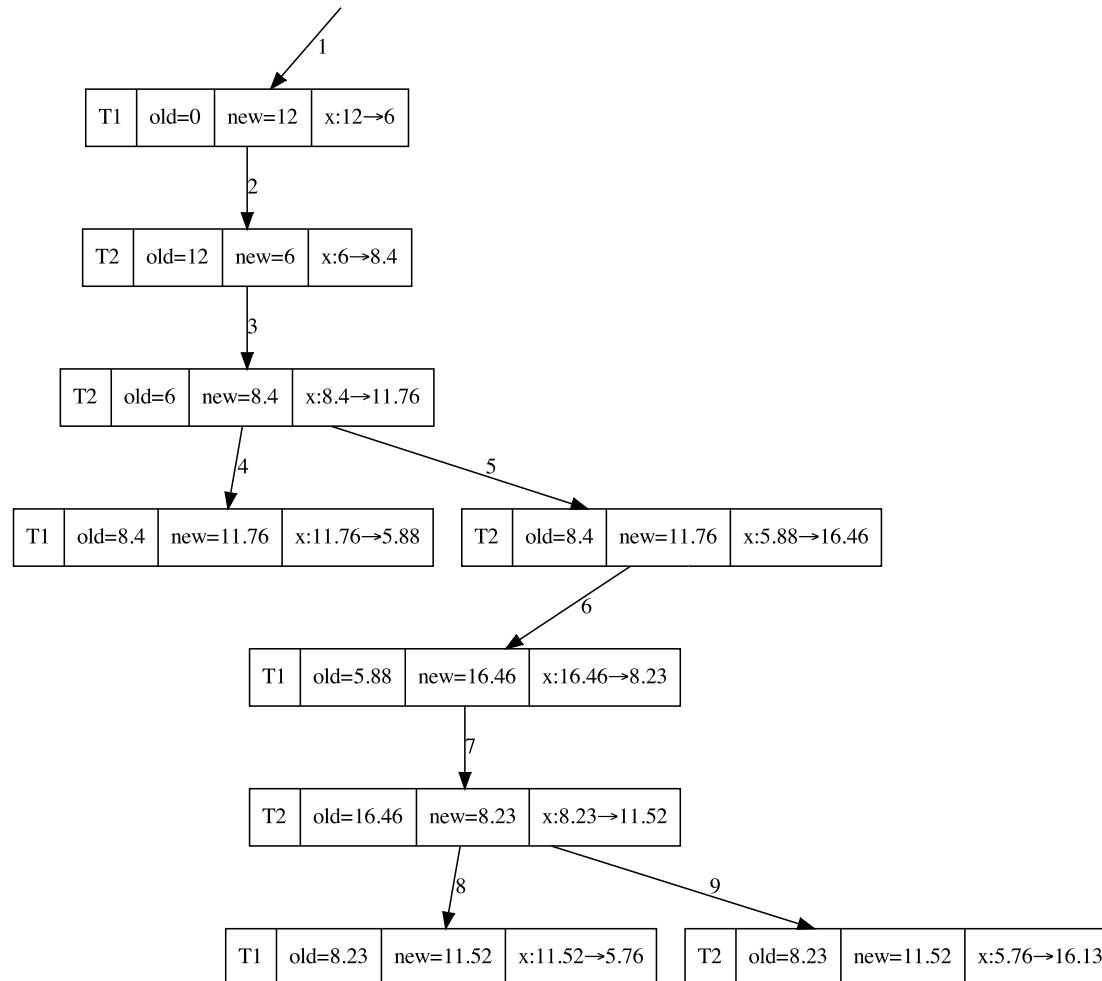
- ...and on

UPDATE ttest SET x = 12;



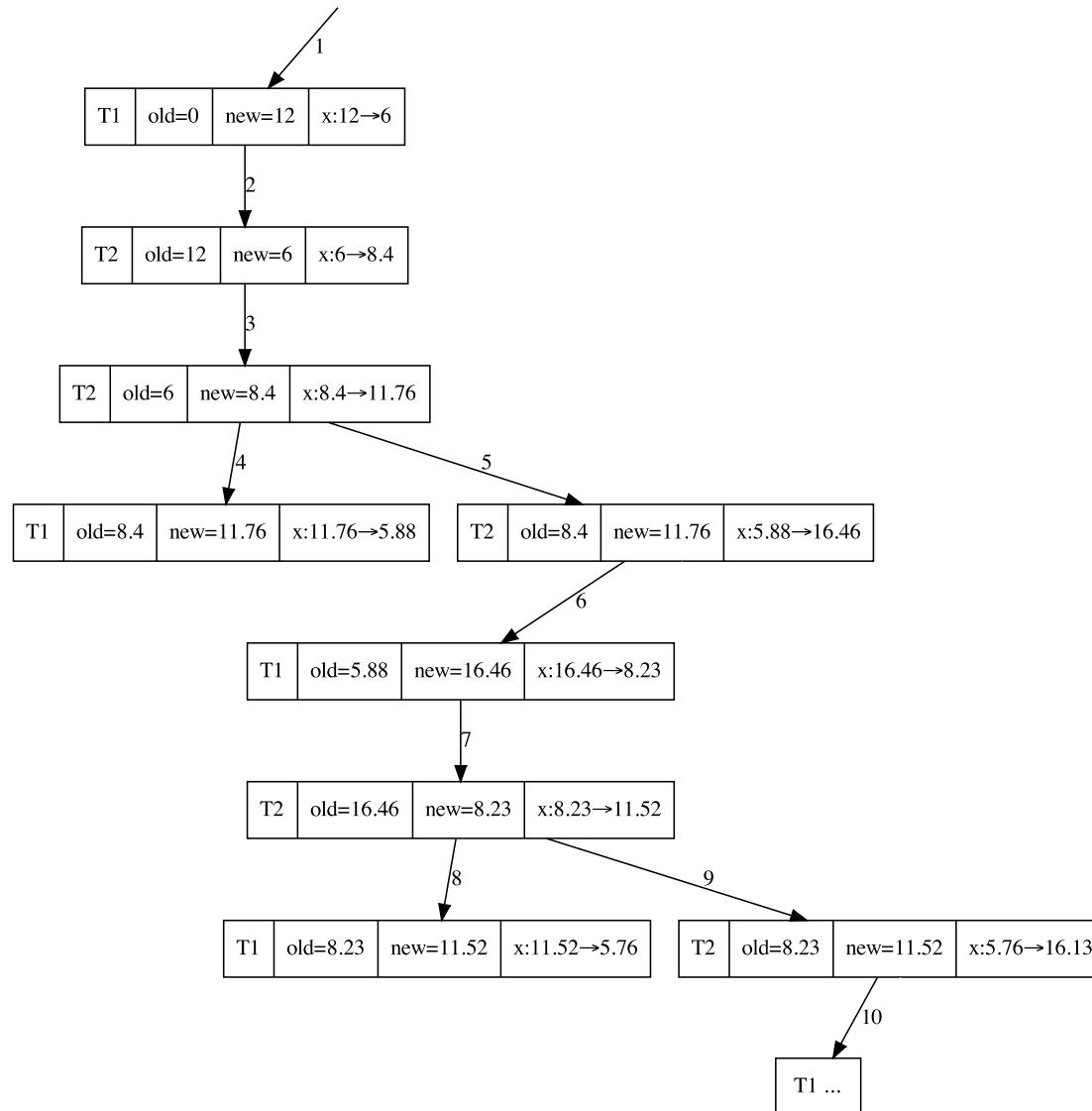
- ...and on

UPDATE ttest SET x = 12;



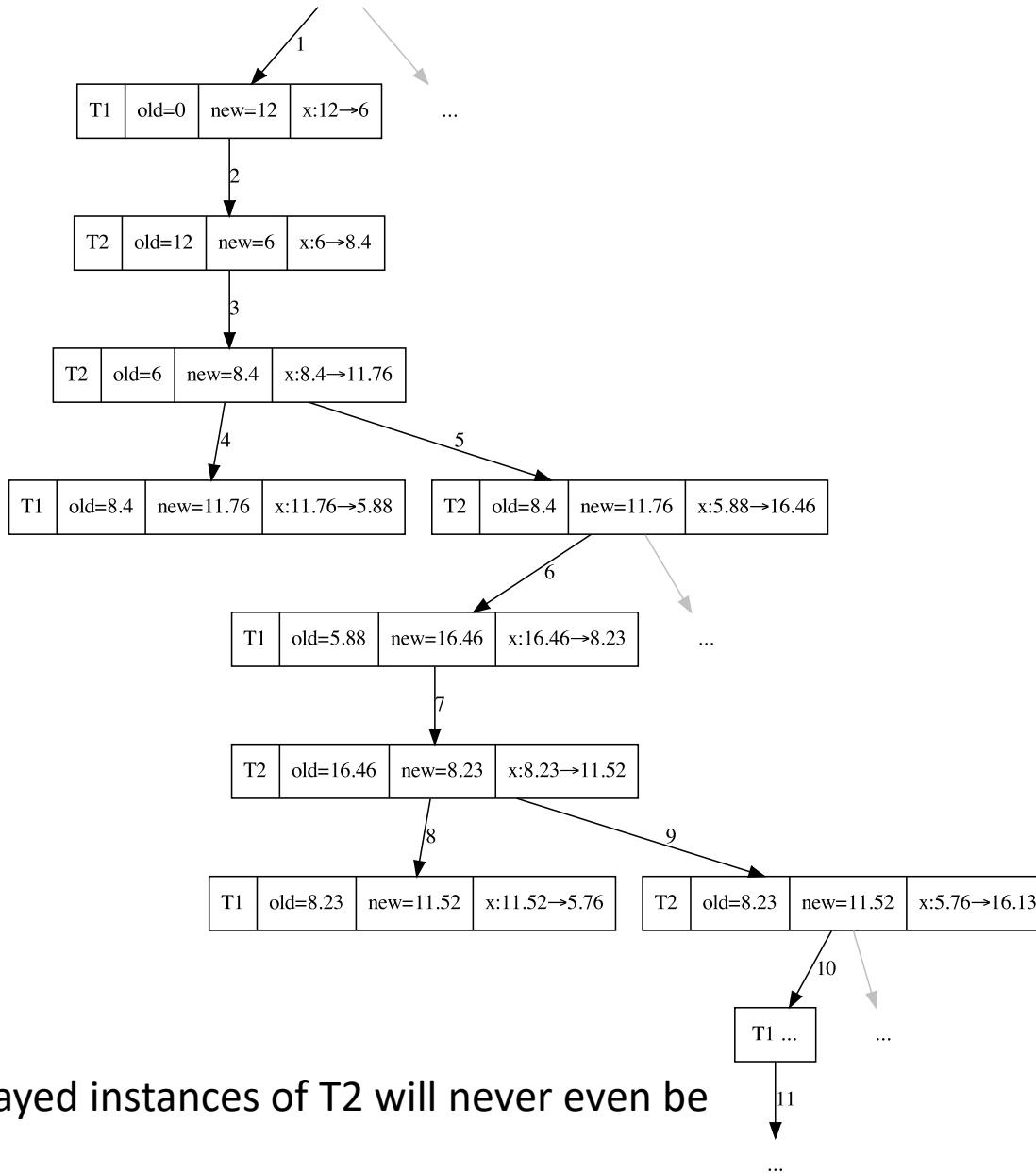
- ...and when the value of x becomes less than 6, we branch to execute a delayed instance of T2

UPDATE ttest SET x = 12;



- Here, the process never stops
- Eventually, the system will report an error

UPDATE ttest SET x = 12;



- Older, delayed instances of T2 will never even be executed

Trigger design principles

- Use triggers to guarantee that when a specific operation is performed, related actions are performed
- Do not define triggers that duplicate features already built into the DBMS. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints
- Limit the size of triggers. If the logic for your trigger requires much more than 60 lines of code, it is better to include most of the code in a stored procedure and call the procedure from the trigger
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- Avoid recursive triggers if not absolutely necessary. Trigger may fire recursively until the DBMS runs out of memory.
- Use triggers judiciously. They are executed for every user every time the event occurs on which the trigger is created
 - https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_tr.htm

Evolution of active databases

- Execution mode (immediate, deferred, detached)
- New events (system-defined, temporal, user-defined)
- Complex events and event calculus
- Instead-of clause
- Rule administration: priorities, grouping, dynamic activation and deactivation
- Variations introduced by vendors (e.g., Oracle)

Proprietary limitations and extensions: Oracle

- Oracle follows a different syntax (multiple events allowed, no table variables, when clause only legal with row-level triggers)

```
create trigger TriggerName
{ before | after } <event> [, <event> [, <event> ]]
[ [ referencing [old [row] [as] OldTupleName ]
      [ new [row] [as] NewTupleName ] ]
for each row
[when SQLPredicate ]]
PL/SQLStatements
<event> ::= { insert | delete | update [of Column] } on Table
```

- They have also a rather different conflict semantics, and no limitation on the expressive power of the action of before triggers

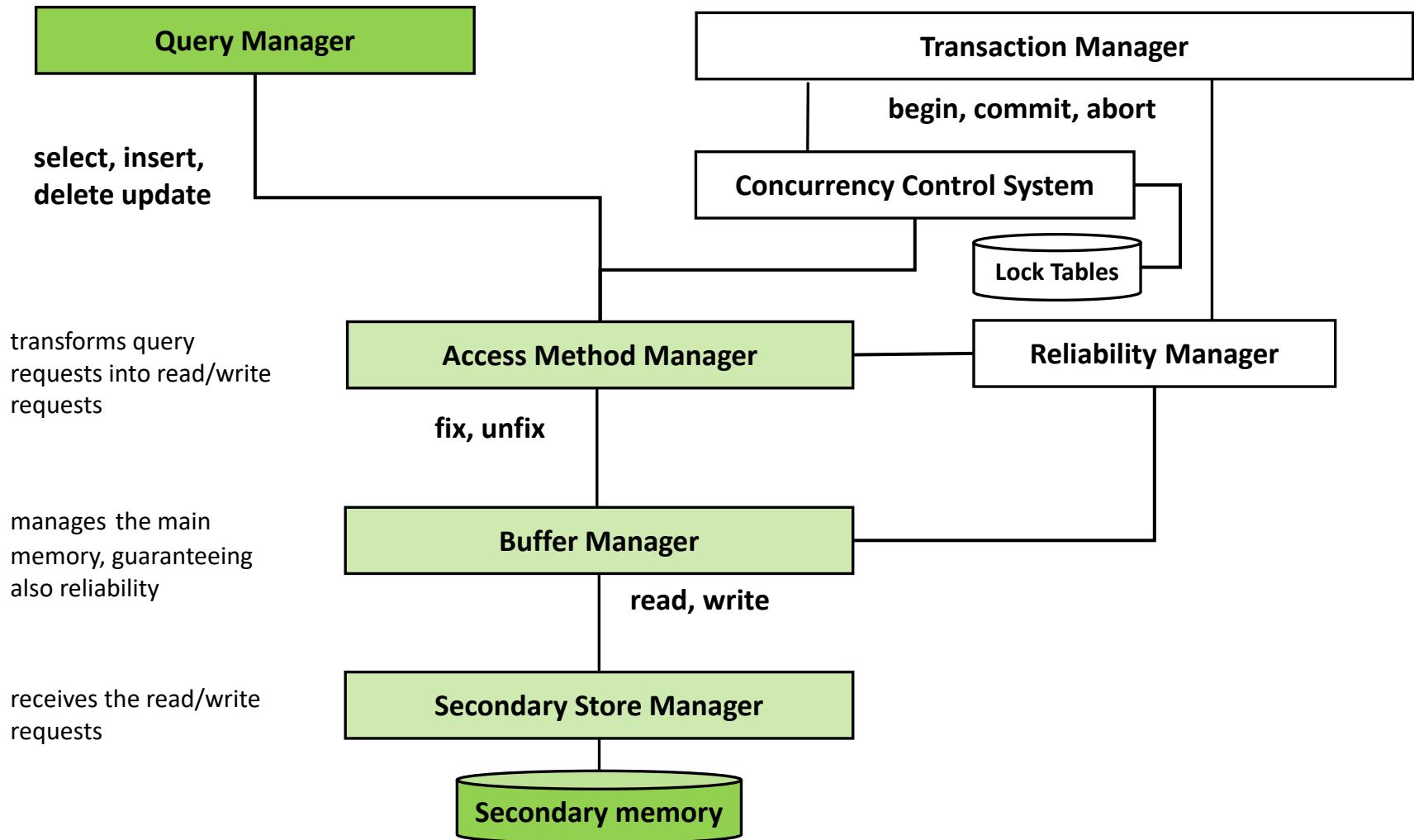
Conclusions

- All major relational DBMS vendors have some support for triggers
- Most products support only a subset of the SQL-99 trigger standard
- Most do not adhere to some of the more subtle details of the execution model
- Some trigger implementations rely on proprietary programming languages
 - portability across different DBMSs is difficult
- Central management of semantics in DB, under the control of the DBMS (not replicated in all the applications)
- To guarantee properties of data that cannot be specified by means of integrity constraints
- Always document them, because their behavior is “hidden”

Databases 2

Physical data structures and query optimization

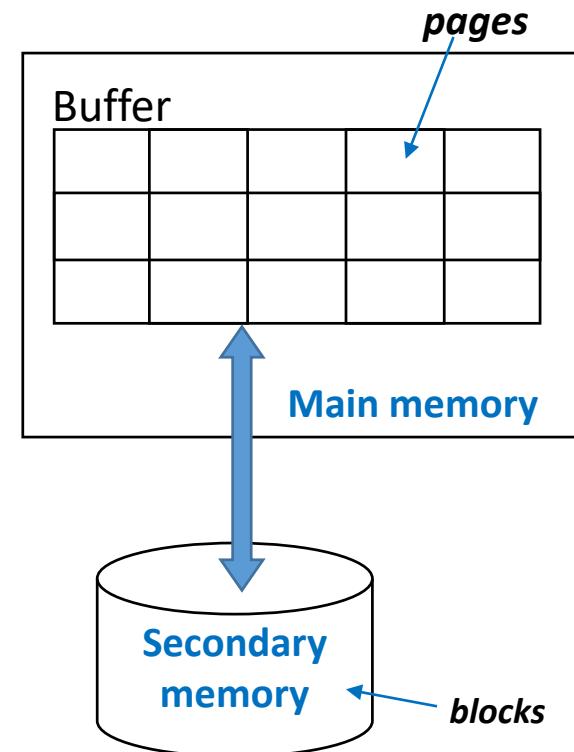
Query management in the DBMS architecture



DATA ACCESS and COST MODEL

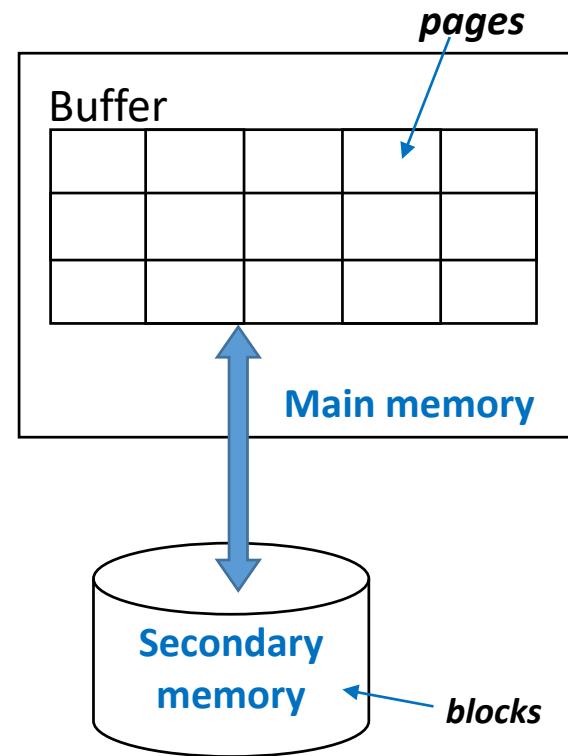
Main and Secondary memory

- Databases must be stored (mainly) in files onto secondary memory for two reasons:
 - size
 - persistence
- Data stored in secondary memory can only be used if first transferred to main memory
- **Page**: unit of storage in main memory
- **Block**: unit of storage in secondary memory



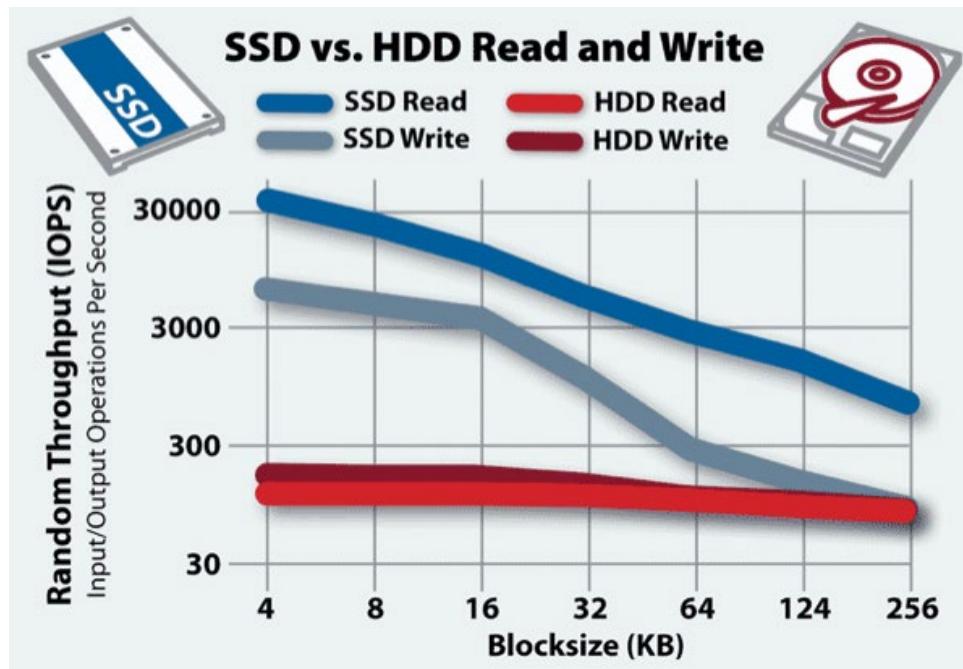
Pages and blocks

- Secondary memory devices are organized in **blocks** of (usually) fixed length
 - order of magnitude: a few KBytes
- We make the *assumption* that the size of a **page** equals the size of a block
 - In real systems a page may contain several blocks
- **I/O operation**: moving a block from secondary to main memory and vice-versa



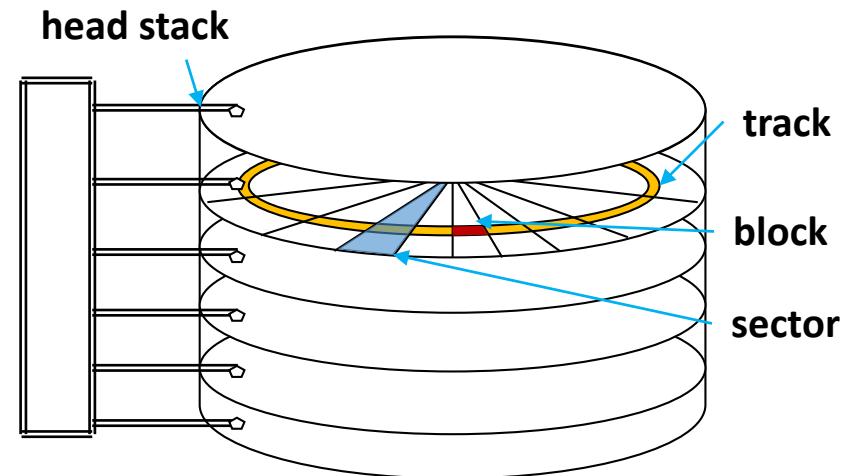
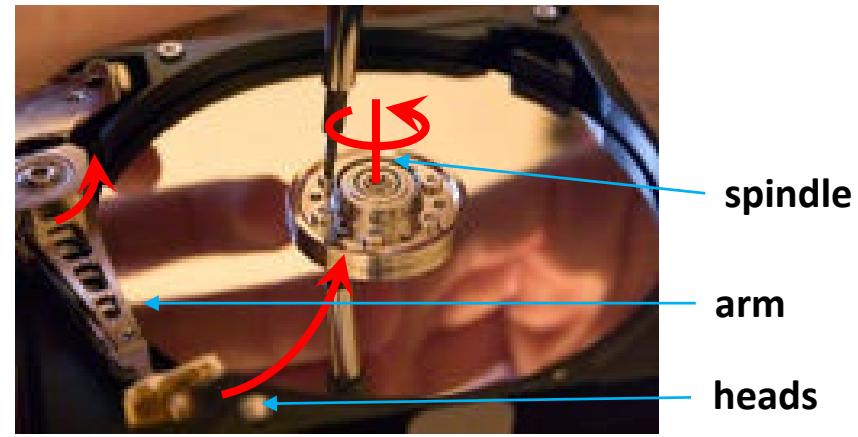
I/O operations

- How long does it take to read a block from a disk?
 - It depends on the technology
 - Mechanical hard drives
 - Solid State Drives (SSD)
 - Fast read/write speed
 - SSD are more expensive
 - For very large datasets, cheap storage is the only viable option



A mechanical hard drive (Winchester)

- Several **disks** piled and rotating at constant angular speed
- A **head stack** mounted onto an arm moves radially in order to reach the **tracks** at varius distances from the rotation axis (**spindle**)
- A particular **sector** is «reached» by waiting for it to pass under one of the heads
- Several **blocks** can be reached at the same time (as many as the number of heads/disks)



→ in depth in the course
Computing Infrastructures, semester 2

Main vs. Secondary memory access time

- Secondary memory access:
 - **seek** time (8-12ms) - head positioning on the correct track
 - **latency** time (2-8ms) - disc rotation on the correct sector
 - **transfer** time (~1ms) - data transfer
- The cost of an access to secondary memory is **4 orders of magnitude** higher than that to main memory
- In "I/O bound" applications the cost exclusively depends on the number of accesses to secondary memory
- **Cost of a query = #blocks** to be moved to execute a query

DBMS and file system

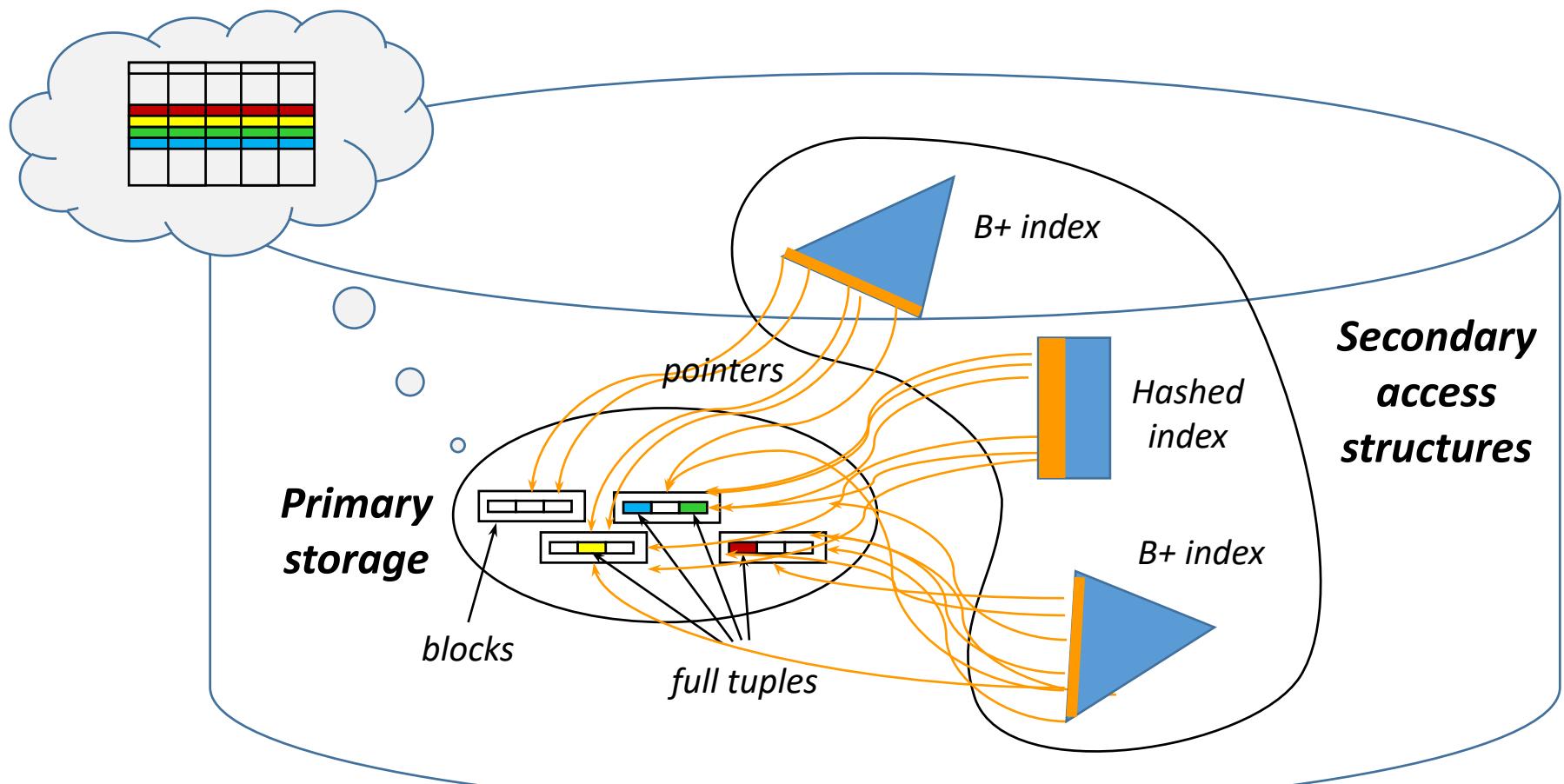
- File System (FS): component of the OS which manages access to secondary memory
- DBMSs make limited use of FS functionalities (create/delete files, read/write blocks)
- The **DBMS directly manages the file organization**, both in terms of the distribution of records within blocks and with respect to the internal structure of each block
 - A DBMS may also control the physical allocation of blocks onto the disk (for faster sequential reads, fine grain control of write operation execution time, reliability, etc)

PHYSICAL ACCESS STRUCTURES

Physical access structures

- Each DBMS has a distinctive and limited set of access methods
 - **Access methods**: software modules that provide data access and manipulation (store and retrieve) primitives **for each physical data structure**
- Access methods have their own data structures to organize data
 - Each table is stored into exactly one **primary** physical data structure, and
 - may have one or many optional **secondary** access structures

Primary and Secondary access structures



Physical access structures

- **Primary structure**: it contains all the **tuples** of a table
 - Main purpose: to store the table content
- **Secondary structures**: are used to **index** primary structures, and only contain the values of some fields, interleaved with pointers to the blocks of the primary structure
 - Main purpose: to speed up the search for specific tuples, according to some search criterion
- Three main types of data access structures:
 - **Sequential** structures
 - **Hash-based** structures
 - **Tree-based** structures

Physical access structures

- Not all types of structures are equally suited for implementing the primary storage or a secondary access method

	Primary	Secondary
Sequential structures	Typical	Not used
Hash-based structures	In some DBMSs (e.g., Oracle hash clusters, IDM DB2 “organize by hash” tables)	Frequent
Tree-based structures	Obsolete/rare	Typical

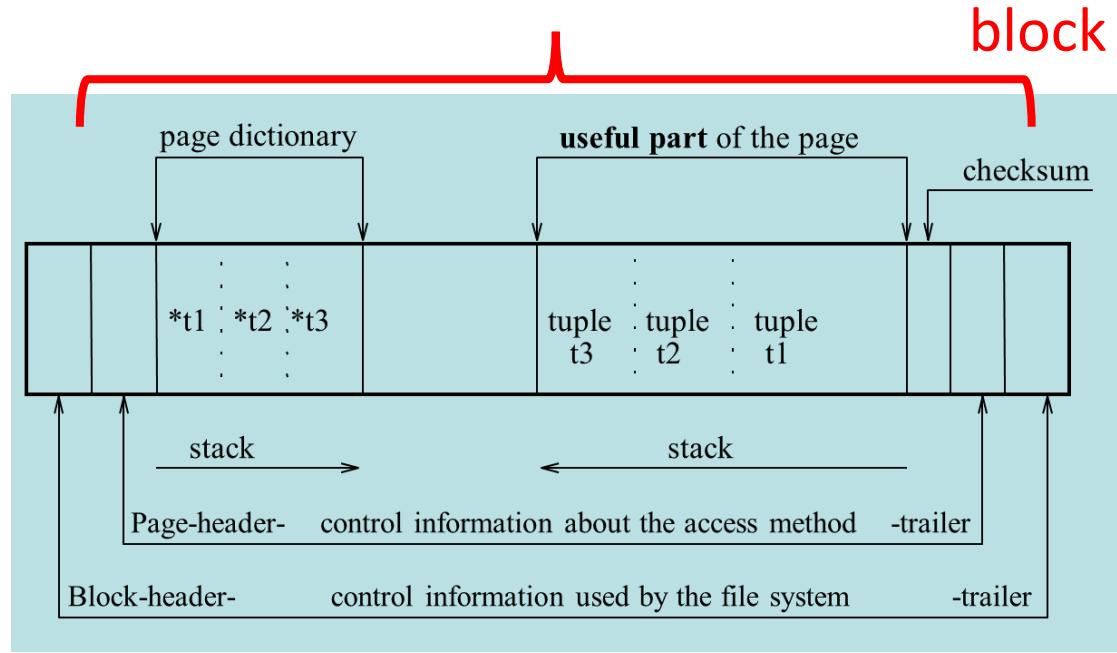
- https://docs.oracle.com/cd/B19306_01/server.102/b14231/hash.htm
- <http://db2portal.blogspot.com/2012/07/db2-hashing-and-hash-organized-tables.html>
- <https://oracle-base.com/articles/8i/index-organized-tables>

Blocks and tuples

- **Blocks**: the "physical" components of files
- **Tuples**: the "logical" components of tables
 - The size of a block is typically **fixed** and depends on the file system and on how the disk is formatted
 - The size of a tuple (also called record) depends on the database design and is typically **variable** within a file
 - optional (possibly null) values, varchar (or other types of non-fixed size) attributes

Organization of tuples in blocks

*Block for sequential
and hash-based methods*



- Block header and trailer with control information used by the file system
- Page header and trailer with control information about the access method
- A page dictionary, which contains pointers (offset table) to each elementary item of useful data (e.g., tuple) contained in the page
- A useful part, which contains the data (e.g., tuples)
 - In general, page dictionaries and useful data grow as stacks in opposite directions
- A checksum, to detect corrupted data

How many tuples in a block: Block Factor

- **Block factor B**: the number of tuples within a block
- Fundamental for **estimating cost of queries**
 - SR: Average size of a record/tuple (assuming "fixed length record")
 - SB: Size of a block
 - if $SB > SR$, there may be many tuples in each block:
 - $B = \lfloor SB / SR \rfloor$ ($\lfloor x \rfloor = \text{floor}(x)$)
- The rest of the space can be
 - Non used: unspanned records
 - Used: tuples spanned between blocks (hung-up tuples/records)

Buffer manager primitives

- Operations are performed in main memory and affect pages
- In our cost model we assume pages of equal size and organization as blocks
- Operations are:
 - Insertion and update of a tuple
 - may require a reorganization of the page or usage of a new page
 - also update to a field may require reorganization (e.g., varchar)
 - Deletion of a tuple
 - often carried out by marking the tuple as ‘invalid’ and triggering later reorganization of page asynchronously
 - Access to a field of a particular tuple
 - identified according to an offset w.r.t. the beginning of the tuple and the length of the field itself (stored in the page dictionary)

Sequential structures

- Sequential arrangement of tuples in the secondary memory (access to next element is supported)
- Blocks can be contiguous on disk or sparse
- Two cases:
 - Entry-sequenced organization: sequence of tuples dictated by their order of entry
 - Sequentially-ordered organization: tuples ordered according to the value of a key (one or more attributes)

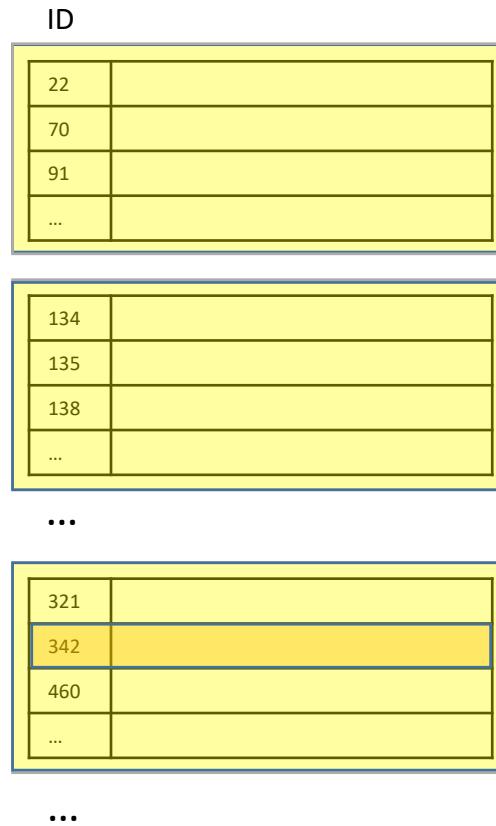
Entry-sequenced sequential (a.k.a. heap) structure

- Effective for
 - Insertion, which does not require shifting
 - Space occupancy, as it uses all the blocks available for files and all the space within the blocks
 - **Sequential** reading and writing (**select * from T**)
 - Especially if the disk blocks are contiguous (seek & latency times reduced)
 - Only if all (or most of) the file is to be accessed
- Non-optimal for
 - Searching specific data units (**select * from T where...**)
 - may require scanning the whole structure
 - But **with indexes** can be used efficiently
 - Updates that increase the size of a tuple (“shifts” required)
 - Shift may require storage in another block
 - Alternative approach: delete old version and insert new one

Sequentially-ordered sequential structure

- Tuples are **sorted based on the value of a key field**
- Effective for
 - Range queries that retrieve tuples having key in an interval
 - Order by and group by queries exploiting the order
- Problem: insertions & updates that increase the size
 - Reordering of the tuples within the block, if space allows
 - Techniques to avoid global reordering:
 - Differential files + periodic merging (example: paper yellow pages)
 - Free space in the block at loading → ‘local reordering’ operations
 - Overflow file: new tuples are inserted into blocks linked to form an overflow chain (general principle used also in other organizations)

Example of query on an entry-sequenced sequential structure



```
SELECT * FROM Student WHERE Student.ID = '342'
```

#I/O operations = # of read blocks

Comparison

	Entry-sequenced	Sequentially-ordered
INSERT	Efficient	Not efficient
UPDATE	Efficient (if data size increases → delete + insert the new version)	Not efficient if data size increases
DELETE	“Invalid”	“Invalid”
TUPLE SIZE	Fixed or variable	Fixed or variable
SELECT * FROM T WHERE key ...	Not efficient	More efficient

Entry-sequenced organization is the most common solution, but PAIRED with secondary access structures

Hash-based access structures

- Efficient **associative** access to data, based on the value of a **key**
- A hash-based structure has N_B **buckets** (with $N_B \ll$ number of data items)
 - A bucket is a unit of storage, typically of the size of 1 block
 - Often buckets are stored adjacent in the file
- A **hash function** maps the key field to a value between 0 and $N_B - 1$
 - This value is interpreted as the index of a bucket in the hash structure (**hash table**)
- Efficient for
 - Tables with small size and (almost) static content
 - Point queries: queries with equality predicates on the key
- Inefficient for
 - Range queries and full table queries
 - Tables with very dynamic content

How hash-based structures work

$\text{hash}(\text{fileId}, \text{Key}) : \text{BlockId}$

- The implementation consists of two parts
 - **folding**, transforms the key values (e.g., strings) so that they become positive integer values, uniformly distributed over a large range
 - **hashing** transforms the positive number into a number between 0 and $N_B - 1$, to identify the right bucket for the tuple

Hash function
 $h(K) = K \bmod 10$

KEY $K=981 \longrightarrow h(981)$: bucket 1

Collisions

- When two keys (tuples) are associated with the same bucket
 - E.g., K=983, K=723 with $h(K)=K \bmod 10 \rightarrow$ bucket 3
- When the maximum number of tuples per block (= bucket capacity) is exceeded, collision resolution techniques are applied:
 - **Closed hashing** (open addressing): try to find a slot in another bucket in the hash table (not used in DBMS)
 - A very simple technique is *linear probing*: visit the next bucket, start again from 0 when you reach the last bucket
 - **Open hashing** (separate chaining):
 - a new bucket is allocated for the same hash result, linked to the previous one

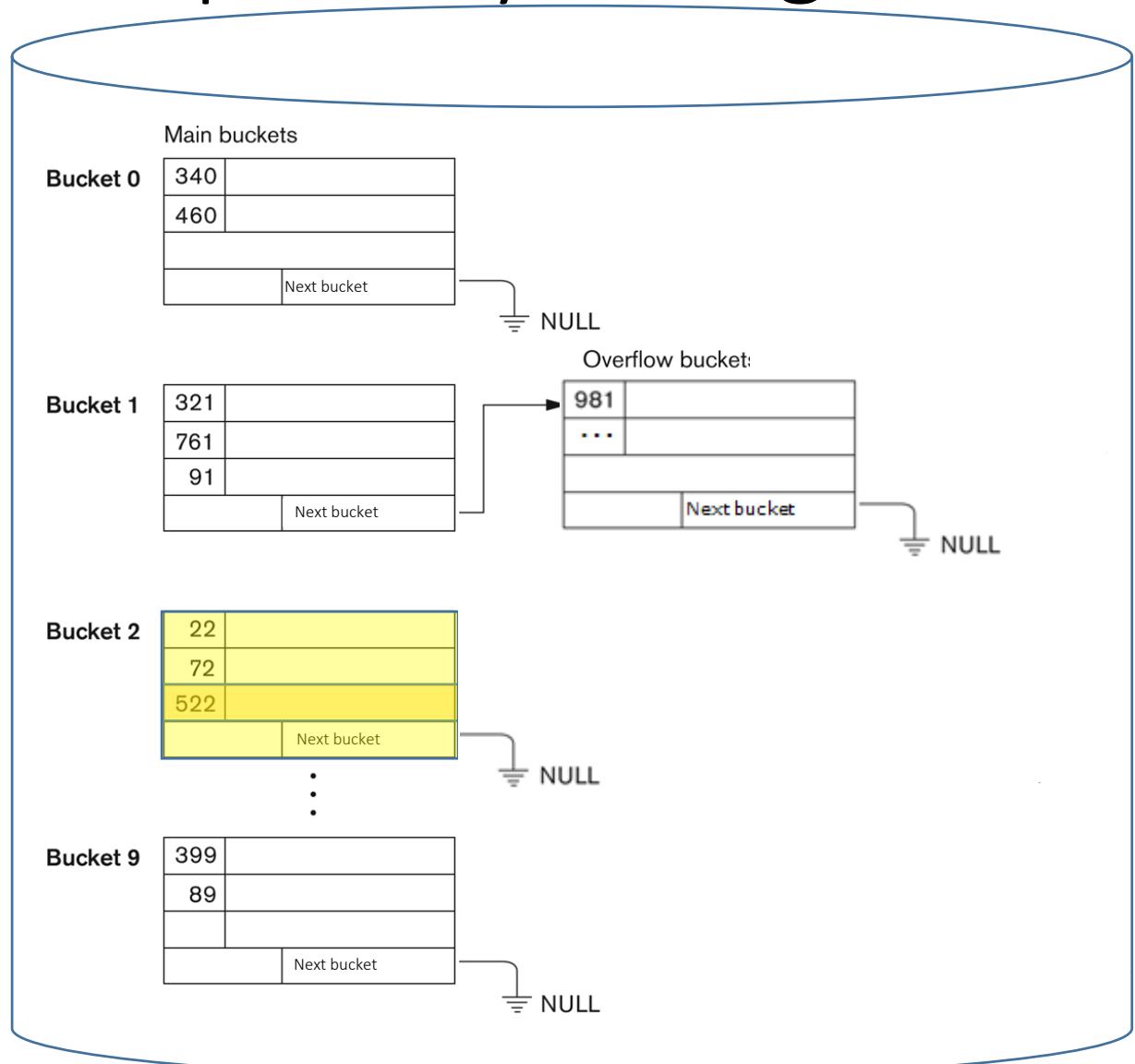
Hash table for primary storage

Hash function
 $h(K) = K \bmod 10$

```
SELECT *
FROM Student
WHERE Student.ID = '522'
```

$$h(522) = 2$$

#I/O operations = 1



Hash table for primary storage

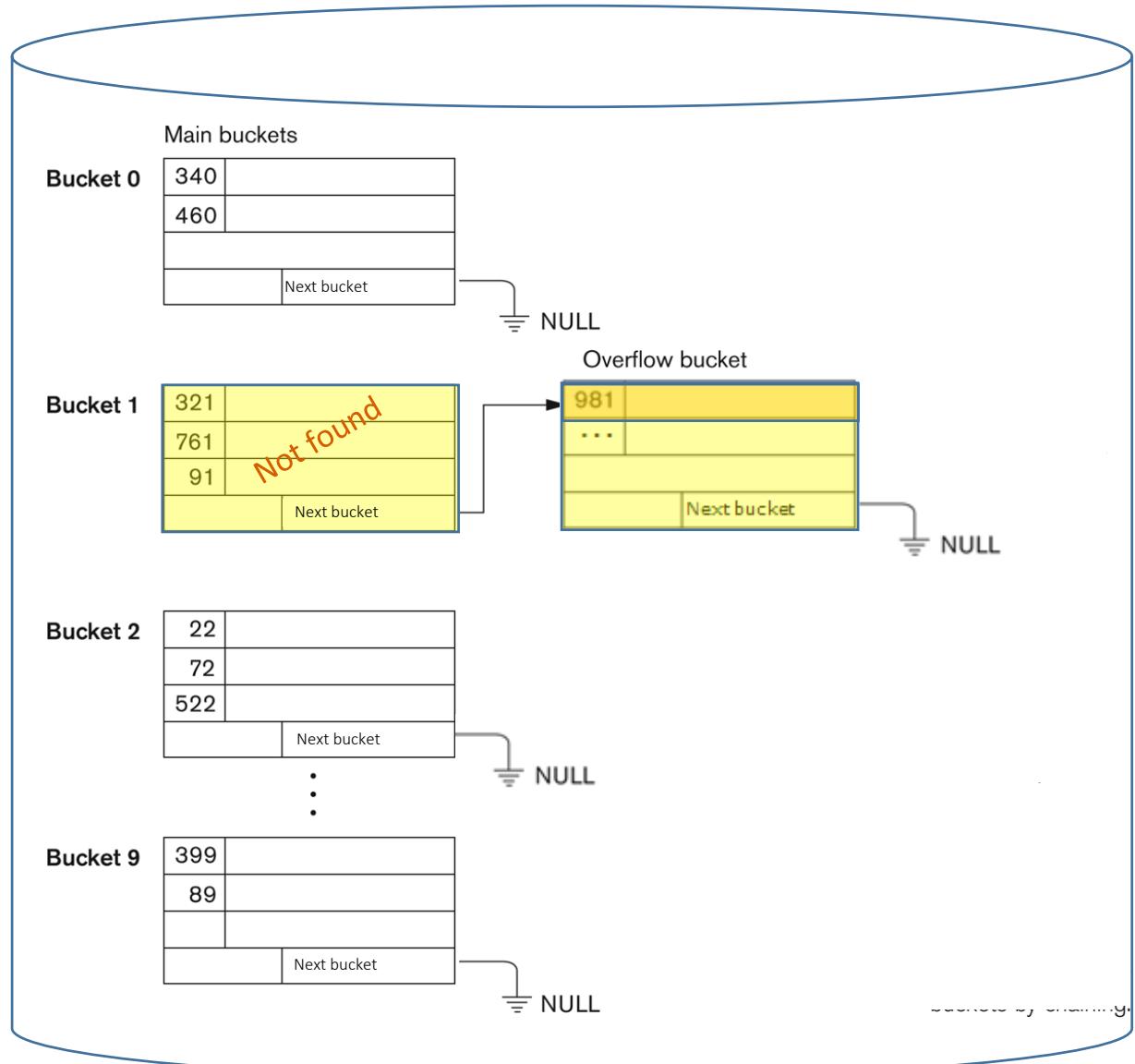
Hash function

$$h(K) = K \bmod 10$$

```
SELECT *
FROM Student
WHERE Student.ID = '981'
```

$$h(981) = 1$$

#I/O operations = 2



Hash-based primary storage - collisions

- How many accesses are needed to find the tuple corresponding to a given key?
- Most of the times 1 access, sometimes 2 accesses or more
- We can estimate the cost of accessing the tuple by considering the average length of the overflow chain

Overflow chains

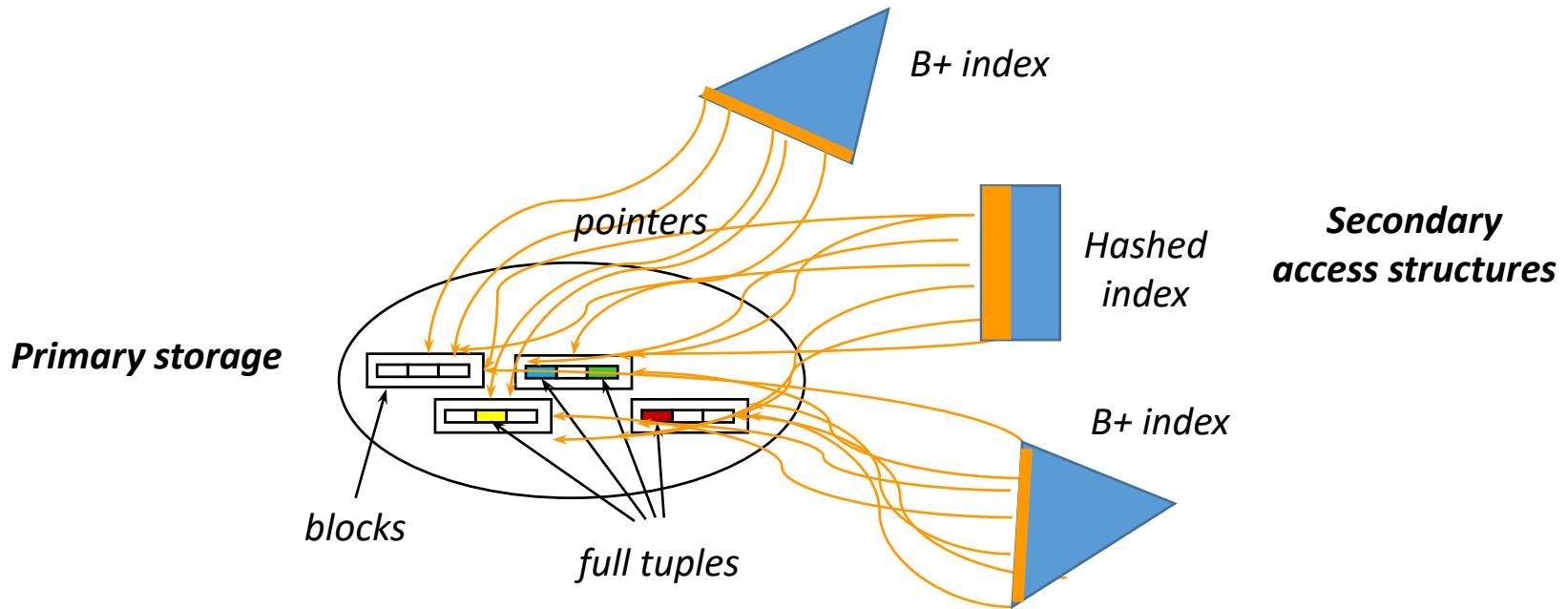
- The average length of the overflow chain is a function of
 - the **load factor** → occupied/available slots → $T / (B \times N_B)$ → average occupancy of a block (%)
 - the **block factor** B , where:
 - T is the number of tuples
 - N_B is the number of buckets
 - B is the number of tuples within a block (1 bucket \leftrightarrow 1 block)
- Example ($B=3$, load factor=70%) → #I/O operations = **1 + 0.3** (with overflow chains)

	1	2	3	5	10	B
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

average # of accesses to the overflow list (computed from real access statistics)

Indexes

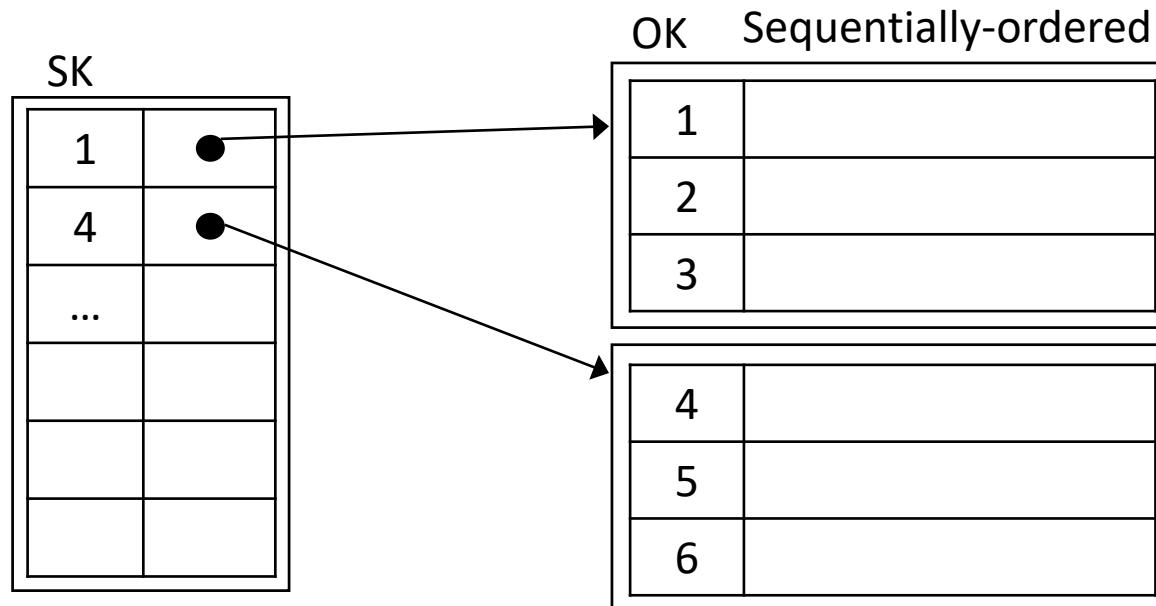
- Data structures that help efficiently retrieve tuples on the basis of a **search key (SK)**
- They contain records of the form [search key, pointer to block]
- Index entries are sorted w.r.t. the key
- The index concept: analytic index of a book → pair (term - page number) alphabetically ordered at the end of a book



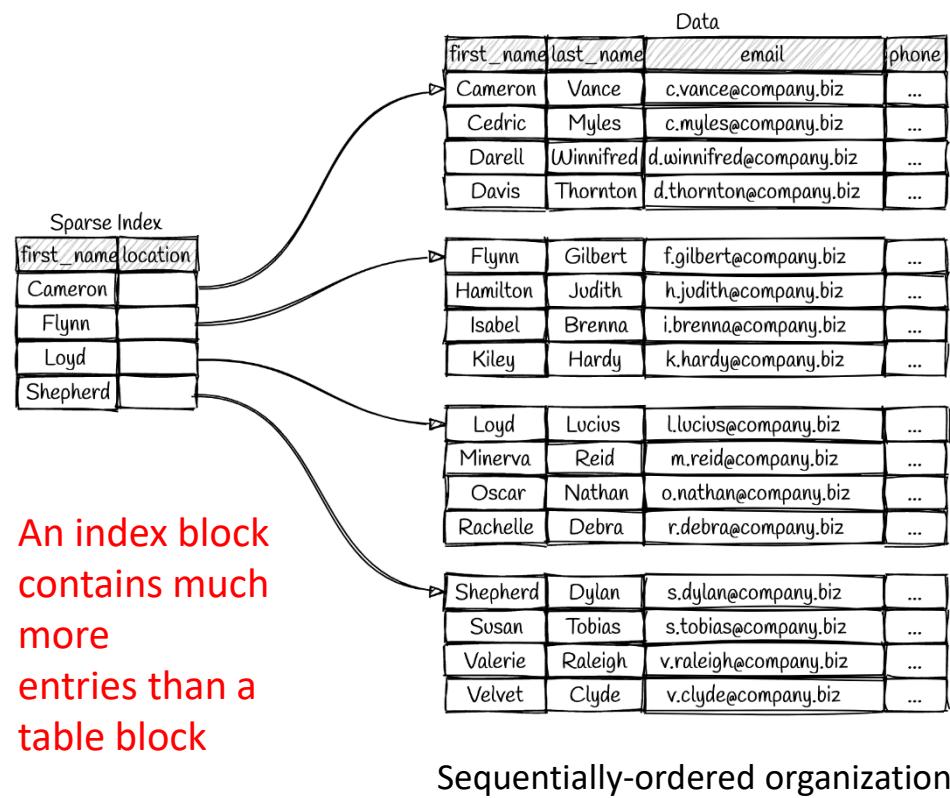
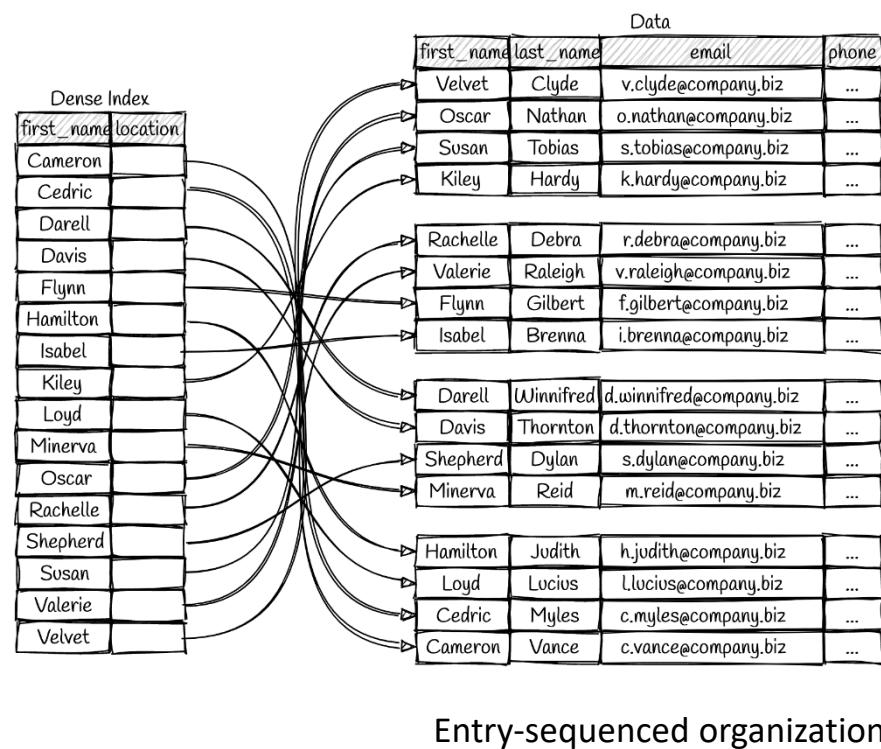
Dense vs. sparse index

- Dense index:
 - An index entry **for each search-key value** in the file
 - Performance is good: only a search on the index is needed + access to the tuple
 - Can be used also on entry-sequenced primary structures
- Sparse index:
 - Index entries **only for some search-key values**
 - Requires less space, but is generally slower in locating the tuple
 - Requires sequentially ordered data structures and **SK = OK (ordering key)**
 - Performance is worse: search on the index + **block scan**
 - Good trade-off: one index entry for each block in the file

Tuple 3 may exist or not
Impossible to know from the index

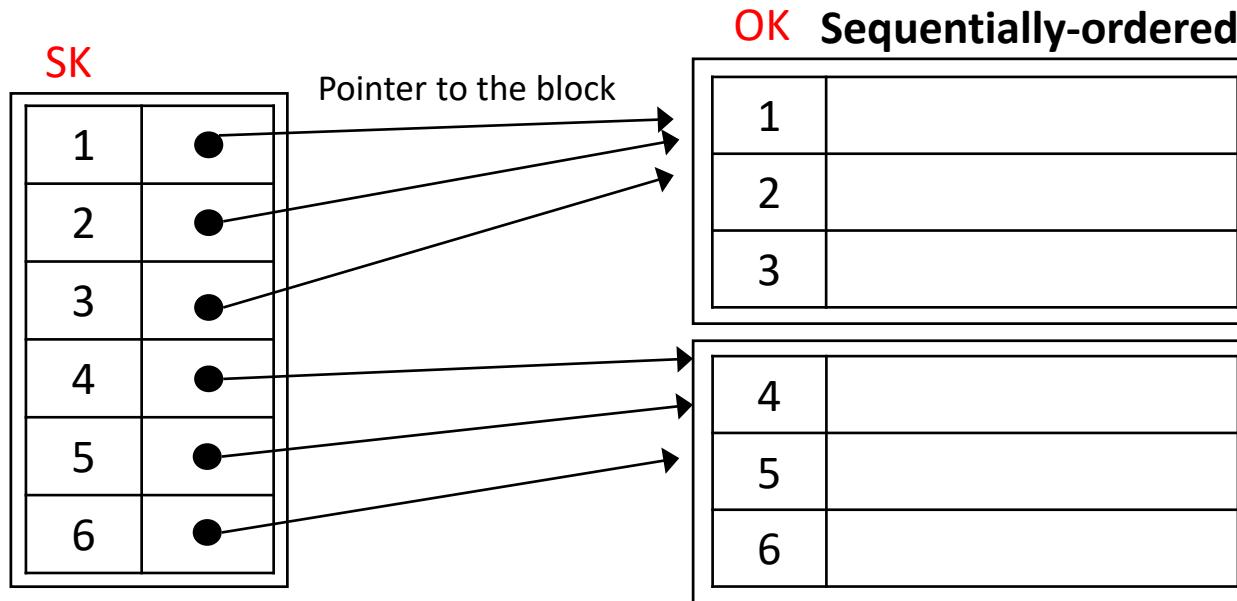


Dense vs Sparse index



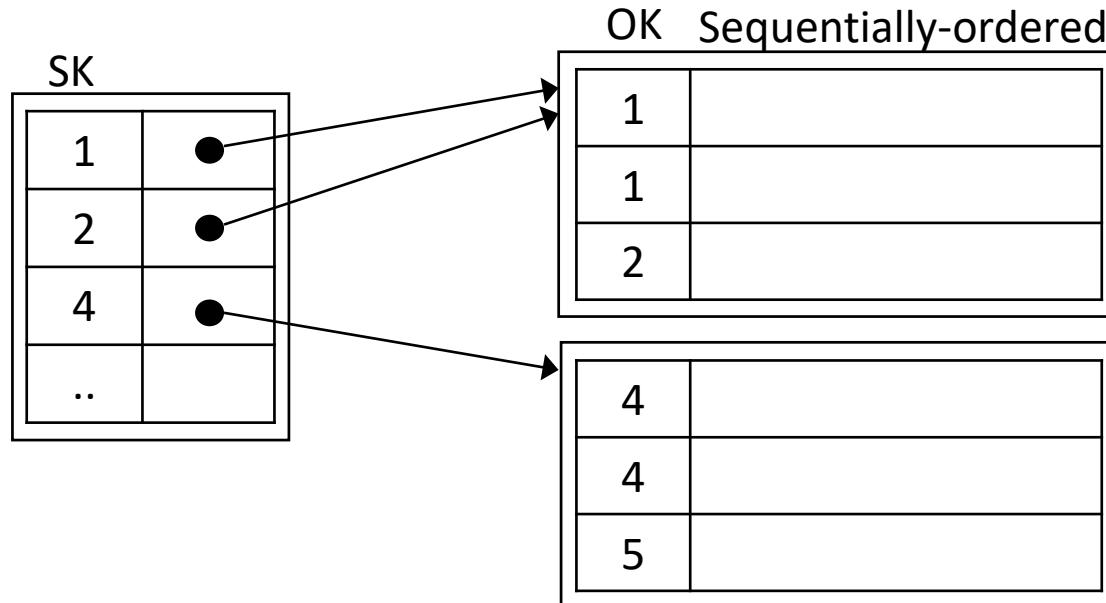
Primary index

- An index defined on **sequentially ordered structures**
- The search key (**SK**) is **unique** and coincides with the attribute according to which the structure is ordered (ordering key - **OK**):
SK = OK
 - **Only one** primary index per table can be defined
 - Usually on the primary key, but not necessarily
 - The index can be dense or sparse



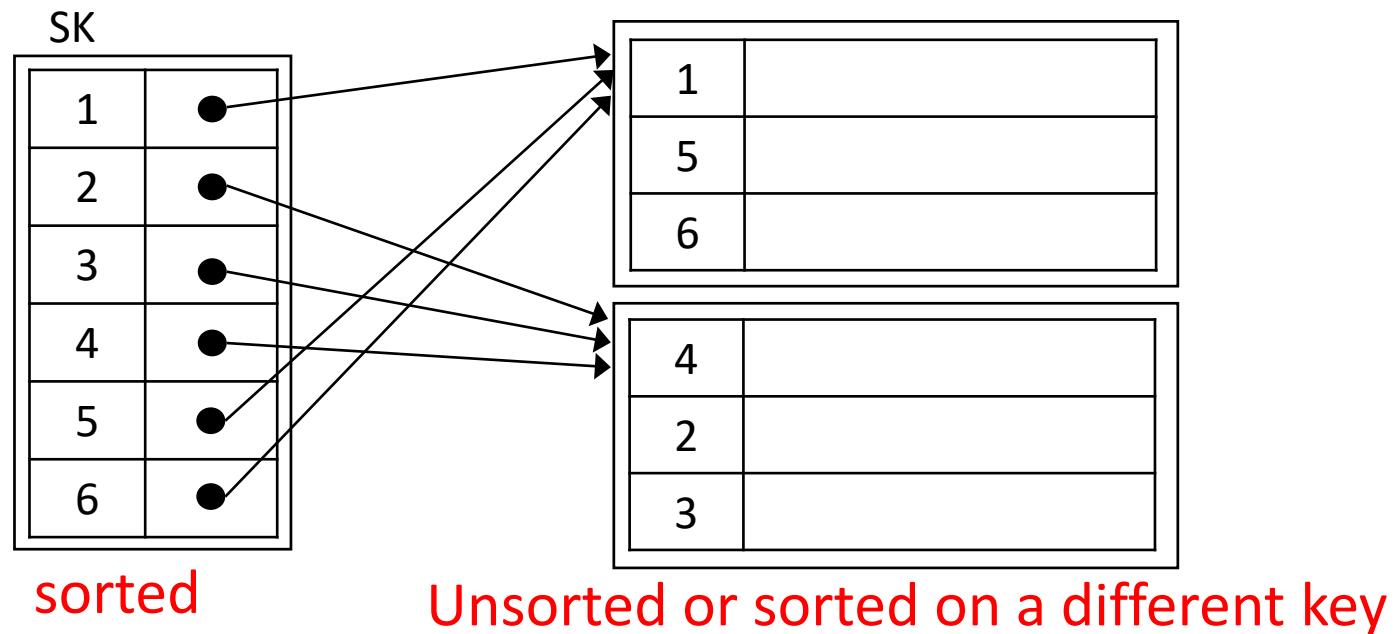
Clustering index

- A generalization of primary index (**SK = OK**) in which the ordering key **can be not unique**
- The pointer of key X refers to the block containing the first tuple of X key
- Could be sparse or dense, but typically sparse: one entry in the index corresponds to multiple tuples



Secondary index

- Secondary index:
 - The search key **specifies an order different from the sequential order of the file ($SK \neq OK$)**
 - Multiple secondary indexes can be defined for the same table, on different search keys
 - It is necessarily **dense**, because tuples with contiguous values of the key can be distributed in different blocks



A search key is not a primary key!

- Don't get confused:
- **Primary key**: set of attributes that uniquely identify a tuple (minimal, unique, not null)
 - Does not imply access path
 - In SQL “PRIMARY KEY” defines a **constraint**
 - Implemented by means of an index
- **Search key**: set of attributes used in an index to speed up tuple location
 - Physical implementation of access structures
 - Defines a common access path
 - Each search key value is associated with one or more pointers
 - May be unique (one pointer) or not unique (multiple pointers or pointer to block)

Summary

Type of Index	Type of structure	Search Key	Density	How many
Primary	Sequentially ordered with SK = OK	Unique	Dense or sparse	One per table
Secondary	Entry sequenced or Sequentially ordered with SK != OK	Unique and non unique	Dense (not possible to scan primary data structure wrt SK)	Many per table
Clustering	Sequentially ordered with SK = OK	Non unique	Typically sparse	One per table

Warning: the terminology is not univocal and changes system by system. Check the meaning of such terms as primary, secondary, key, clustering/clustered. They depend on the context and sometimes on the system

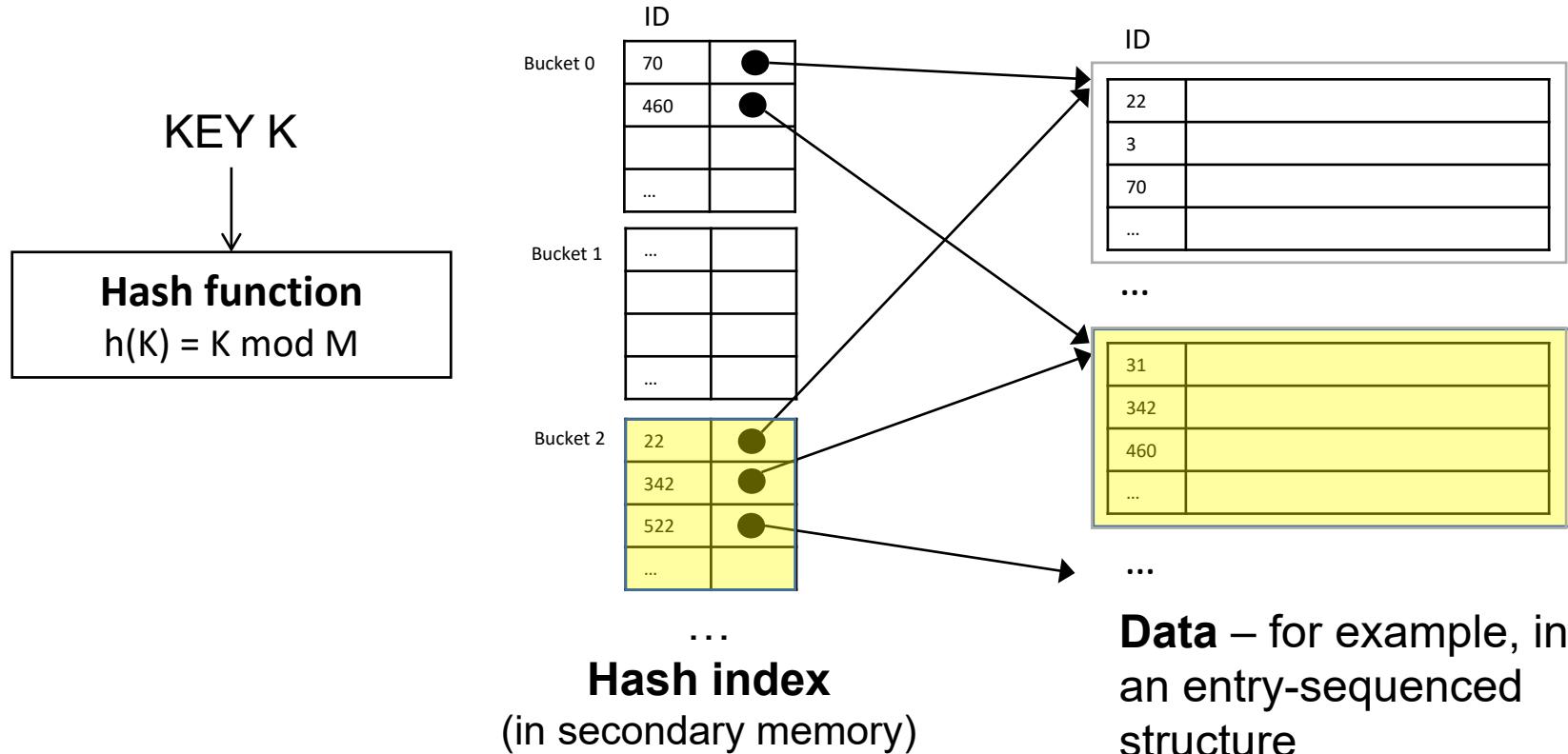
Pros & cons of indexes

- Smaller than primary data structures, can be loaded in main memory
- Support point/range queries and sorted scans efficiently
 - But less efficient than hash primary structures for point queries
- BUT: adding indexes to tables means that the DBMS has to **update the index** too after an insert update or delete operation
- Indexes do not come for free, they may slow down data changing operations

Using hash-based structures as indexes

- Hash-based structures can be used for **secondary** indexes
- Shaped and managed exactly like a hash-based primary structure, but
 - instead of the tuples, **the buckets only contain key values and pointers**

Hash-based index



```
SELECT * FROM Student WHERE Student.ID = '342'
```

#I/O operations = 2 (without overflow chains)

About hashing

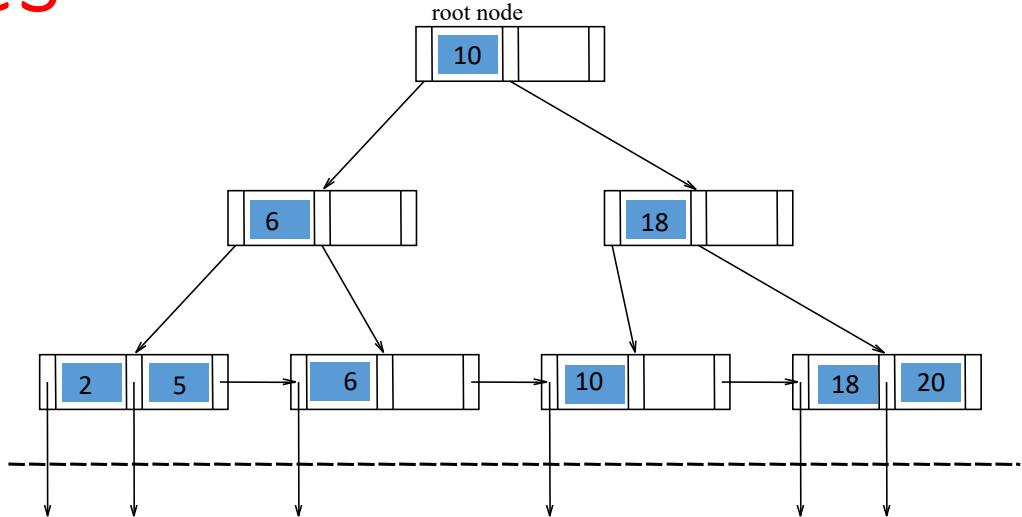
- In a huge database, it is inefficient to search all the index values and reach the desired data
- Hashing can be used to calculate the direct location of an index entry, or even of a data tuple if used as primary storage structure
- Good performance for **equality predicates** on the key field
- Inefficient for access based on
 - interval predicates or
 - the value of non-search-key attributes

Tree-based structures

- Most frequently used in relational DBMSs for secondary (index) structures
 - SQL indexes are implemented in this way
- Support associative access based on the value of a key search field
- **Balanced trees (B trees)**: the lengths of the paths from the root node to the leaf nodes are all equal
 - Balancing improves performance
- Two main file organizations:
 - **B trees**: key values stored *also in internal nodes*
 - **B+ trees** (most used): all key values stored *in leaf nodes*

B+ tree structures

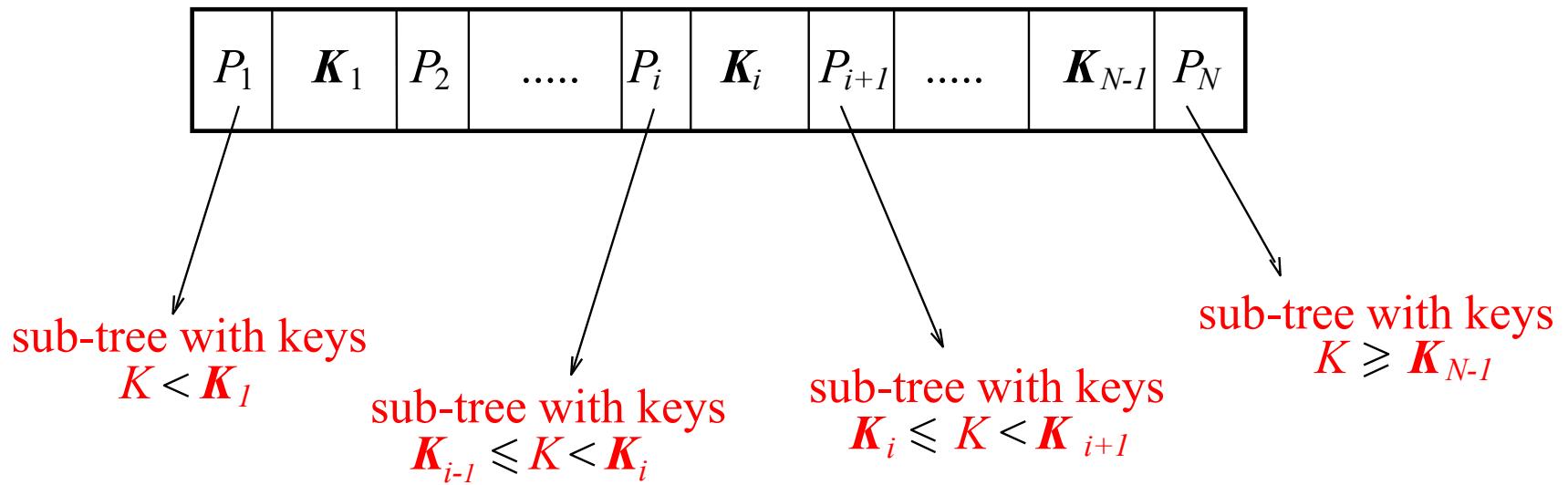
- Evolution from B trees
- Multi-level index
 - one root node
 - several intermediate nodes
 - several leaf nodes
- Each node is stored in a block
- In general, each node has a large number of descendants (fan out or **degree**), and therefore the majority of blocks are leaf nodes
- The fan out depends on the size of the block, the size of key values and the size of pointers



[Examples of insert/delete in a B+ tree \(visualization tool\)](#)

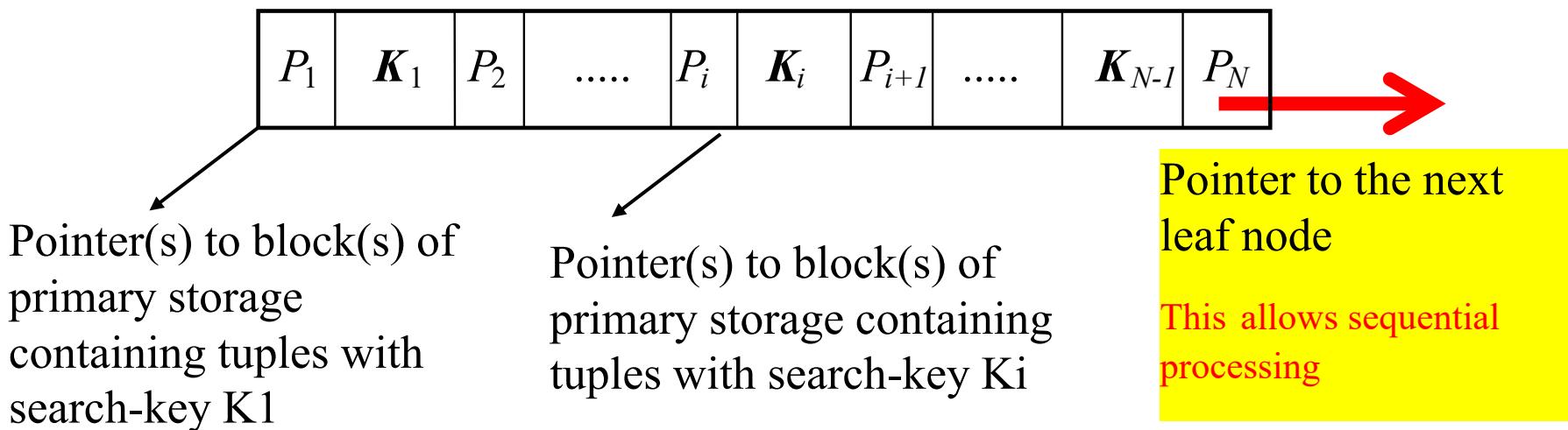
Structure of the B+ tree internal nodes

- Internal nodes form a multilevel (**sparse**) index on the leaf nodes
- Each node can hold up to $N-1$ search-keys and N pointers
- At least $[N/2]$ pointers (except for the root node) are maintained to ensure that each internal node is at least half full
- Search-keys in a node are sorted $K_i < K_j$ with $i < j$



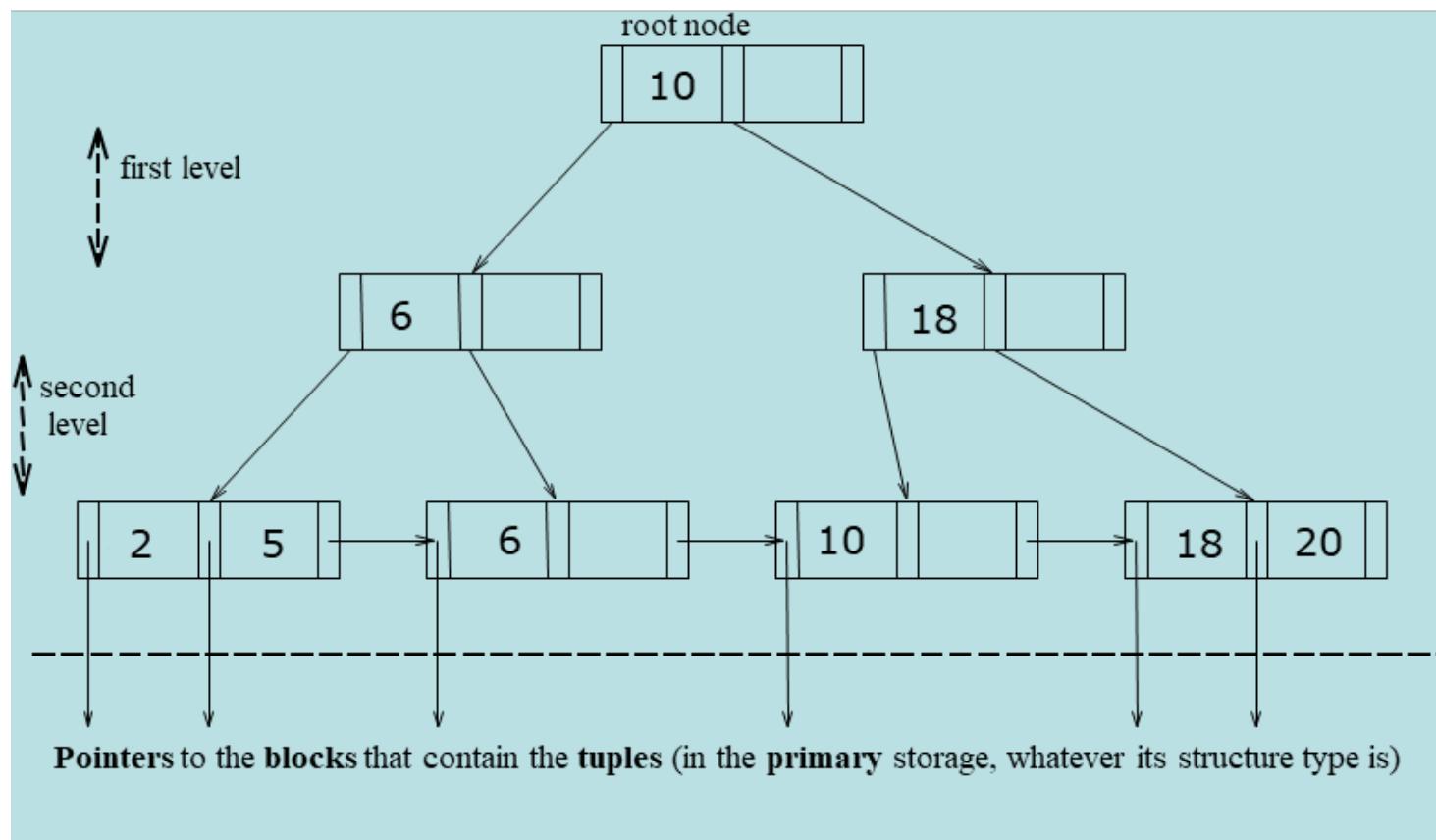
Structure of the B+ tree leaf nodes

- Each node can hold up to N-1 search-keys
- At least $\lceil (N-1)/2 \rceil$ key values should be present in each node (condition ensured by proper maintenance algorithms)
- Search-keys in a node are sorted $K_i < K_j$ with $i < j$
- The set of leaf nodes forms a **dense** index (each existing key value appears in a node)



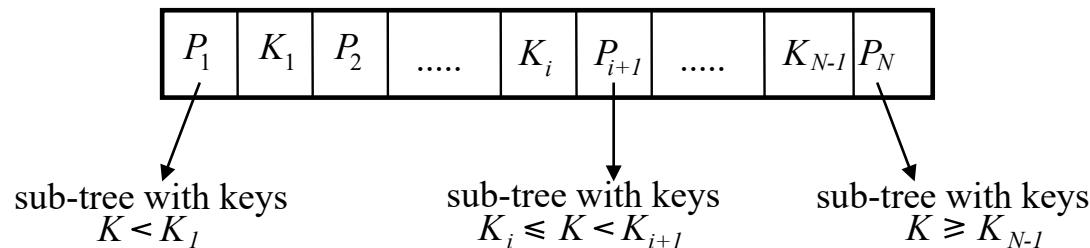
An example of secondary B+ tree

- The leaf nodes are linked in a chain ordered by the key
- Supports interval queries efficiently



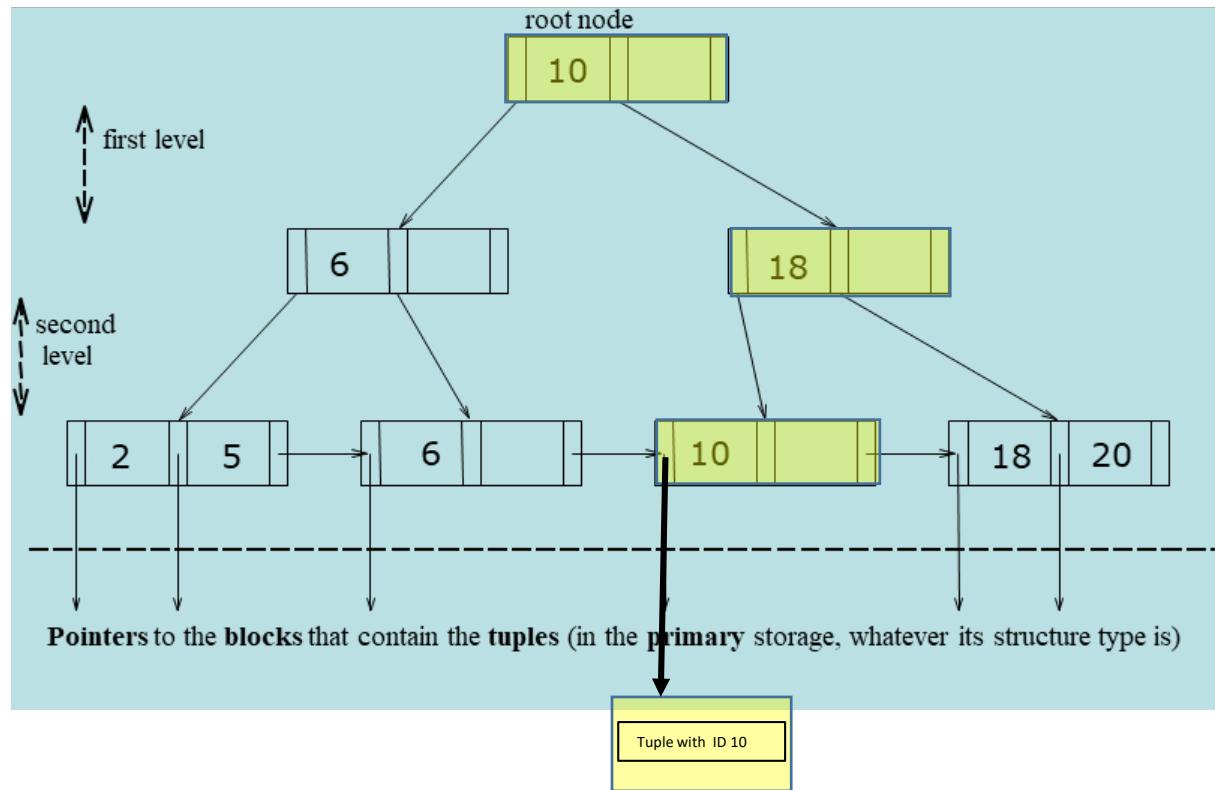
Search technique / lookup

- Looking for a tuple with key value V
- At each intermediate node:
 - if $V < K_1$ follow P_1
 - if $V \geq K_{N-1}$ follow P_N
 - otherwise, follow P_{j+1} such that $K_j \leq V < K_{j+1}$



B+ tree index: query with equality predicate

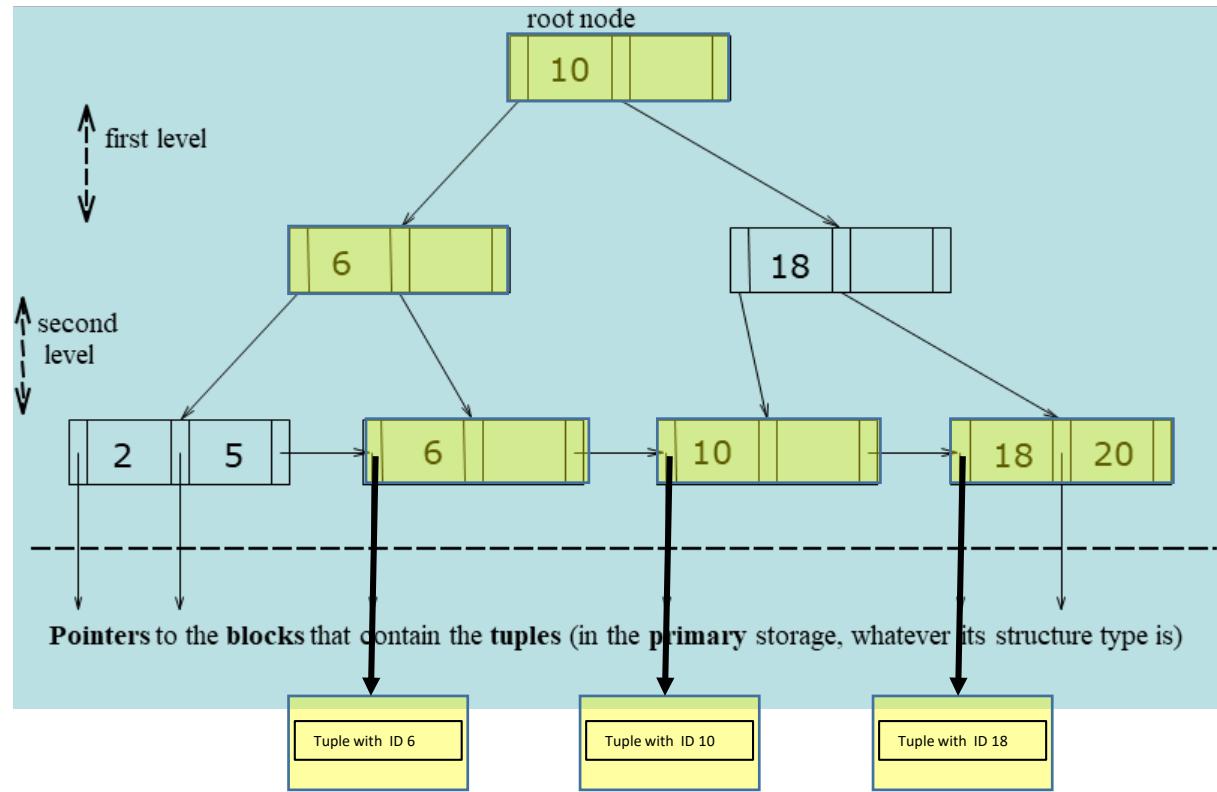
```
SELECT *
FROM Student
WHERE Student.ID = '10'
```



- #I/O operations = **#blocks read to reach the leaf + 1 access to the block containing the tuple**
- In the example: 2 intermediate nodes + 1 leaf node + 1 data block = 4 I/O operations

B+ tree index: query with interval predicate

```
SELECT *
FROM Student
WHERE Student.ID
BETWEEN '6' and '19'
```

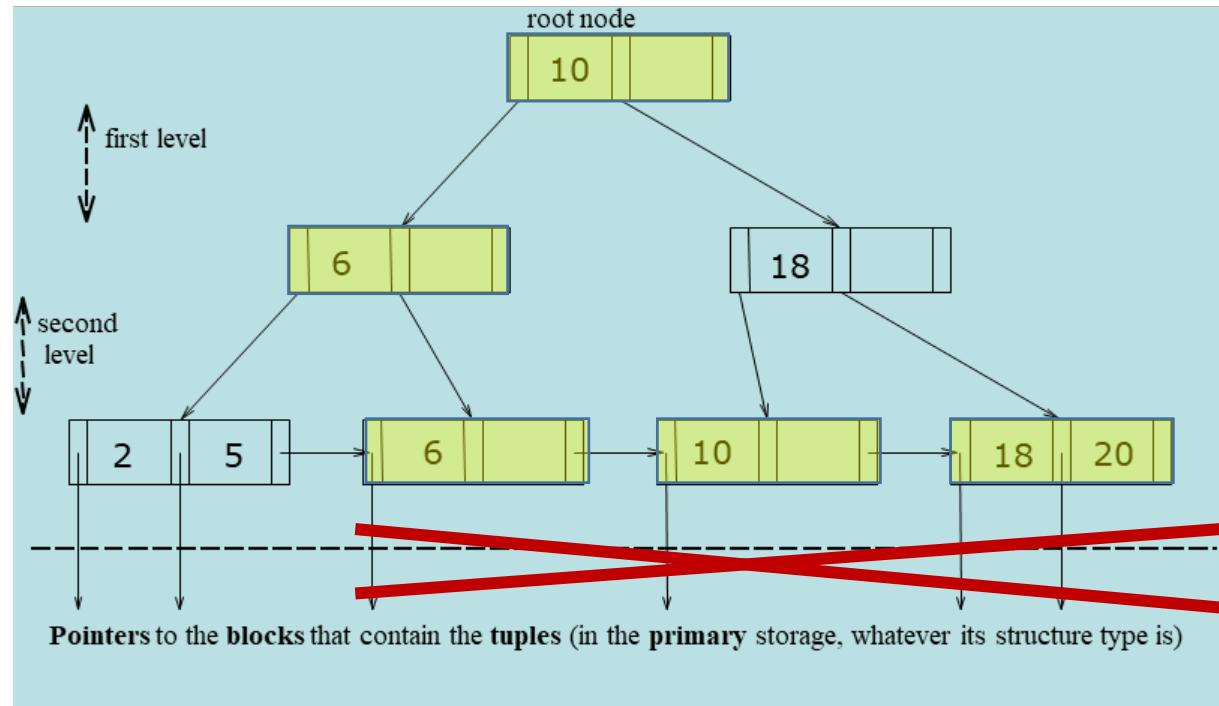


- #I/O operations = #blocks to reach the leaf + # of leaf nodes that are visited + #accesses to the blocks containing the tuples
- In the example: 2 intermediate nodes + 3 leaf nodes + 3 data blocks = 8 I/O operations

B+ tree index: query with interval predicate

```
SELECT Student.ID
```

```
FROM Student  
WHERE Student.ID  
BETWEEN '6' and '19'
```



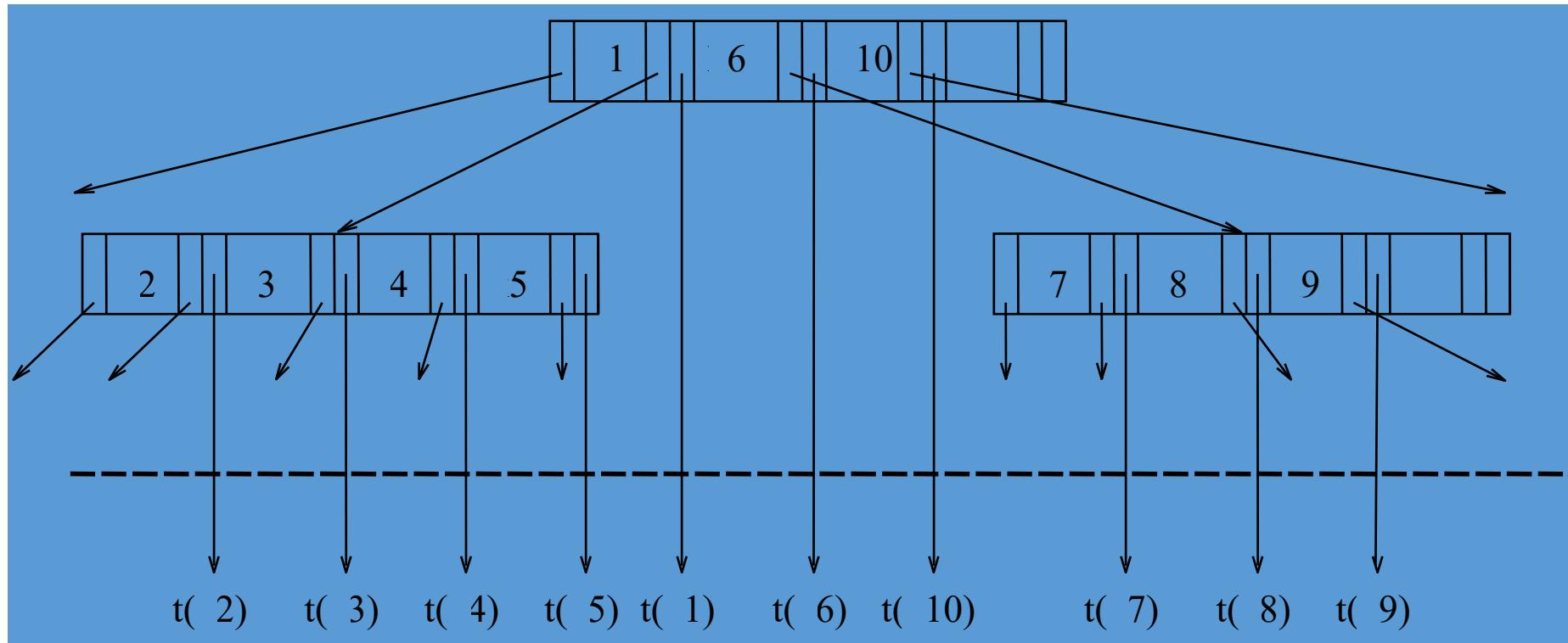
- The leaf nodes of the index contain **all the ID values (dense index)** sorted in ascending order!
- We can access only the B+ tree
- #I/O operations = **#levels to reach the leaf + # of leaf nodes that are visited = 5 I/O**

B-tree

- Eliminates the redundant storage of search-key values
 - Search key values appear only once
 - Intermediate nodes for each key value K_i have:
 - One pointer to the sub-tree with keys between K_i and K_{i+1}
 - One (or more) pointer(s) to the block(s) that contain(s) the tuple(s) that have value K_i for the key
 - (there can be more than one tuple if the key is not unique)
- Lookup can be slightly faster, but interval queries are less efficient

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

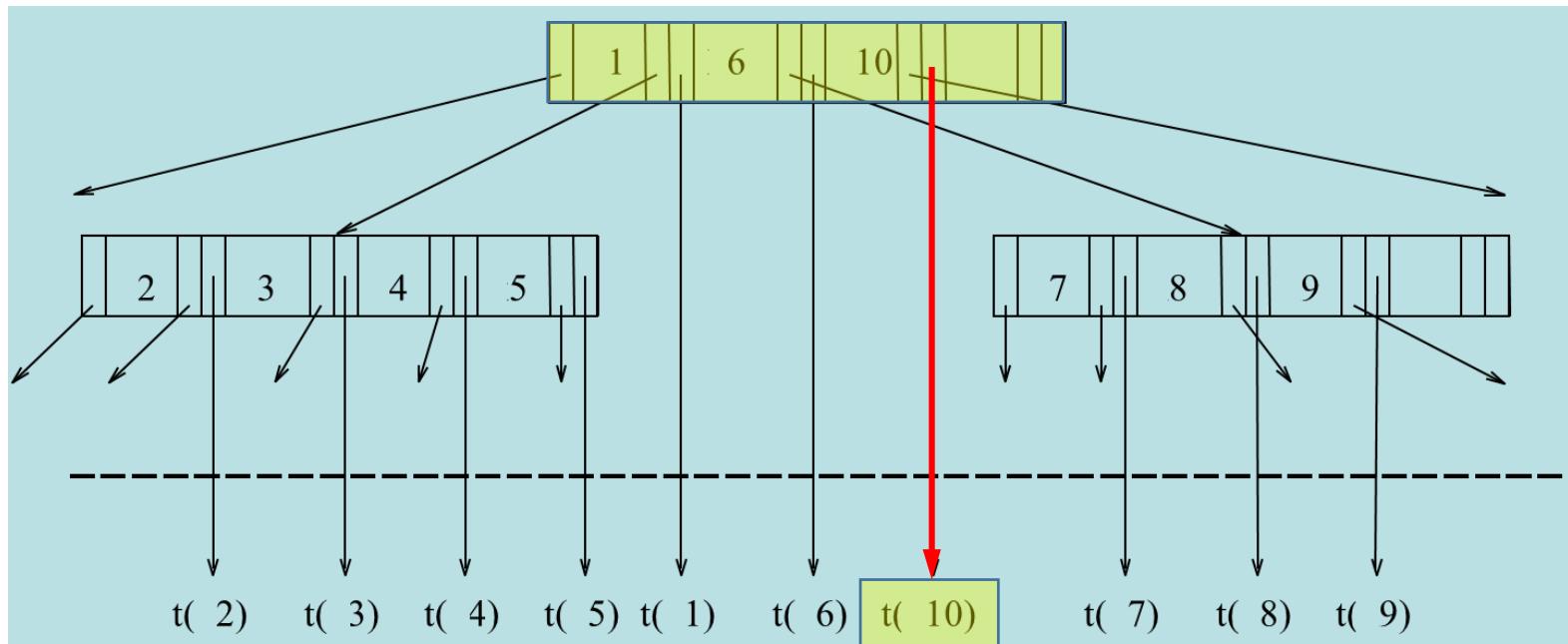
An example of secondary B tree



The leaf nodes form a sparse index and are not linked in a sequential list

Query on B tree

```
SELECT * FROM Student  
WHERE Student.ID='10'
```



- #I/O operations = # levels to find the ID in the tree + 1 access to the block containing the tuple
- In the example: 2 I/O operations

Indexes in SQL

- Syntax in SQL:

```
create [unique] index IndexName on  
TableName(AttributeList)
```

```
drop index IndexName
```

- Some examples:

- [Oracle create index](#)
- [MySQL create index](#)

Indexes in SQL

- Every table should have:
 - A suitable primary storage, possibly sequentially ordered (normally on unique values, typically the primary key)
 - Several secondary indexes, both unique and not unique, on the attributes most used for selections and joins
- Secondary structures are progressively added, checking that the system actually uses them

Some guidelines for choosing indexes

1. Do not index small tables
2. Index Primary Key of a table if it is not a key of the primary file organization
 - Some DBMS automatically create unique indexes on primary keys and unique keys
3. Add a secondary index to any column that is heavily used as a secondary key
4. Add a secondary index to a Foreign Key if it is frequently accessed
5. Add secondary indexes on columns that are involved in: selection or join criteria; ORDER BY; GROUP BY; other operations involving sorting (such as UNION or DISTINCT)
6. Avoid indexing a column or table that is frequently updated
7. Avoid indexing a column if most queries retrieve a significant proportion of the records in the table
8. Avoid indexing columns that consist of long character strings

Databases 2

INTRODUCTION TO OPTIMIZATION

COSTS OF DIFFERENT ACCESS MODES

Query optimization

- We learned about tree & hash indexes
 - How does the DBMS know when to use them?
- The same query can be executed by the DBMS in many ways
 - How does the DBMS decide which option is best?

Query optimization

- Optimizer:
 - it receives a query written in SQL and
 - produces a program in an internal format that uses the data access methods
- Steps:
 - Lexical, syntactic and semantic analysis
 - Translation into an internal representation (similar to algebraic trees)
 - Algebraic optimization
 - Cost-based optimization
 - The query may be “rewritten” by the DBMS, e.g., turning joins into nested queries
 - Code generation

A simple example of query optimization

STUDENT (ID, FirstName, LastName, Email, City,
Birthdate, Sex)

EXAM (SID, CourseID, Date, Grade)

- Query

SELECT LastName, FirstName

FROM Student, Exam

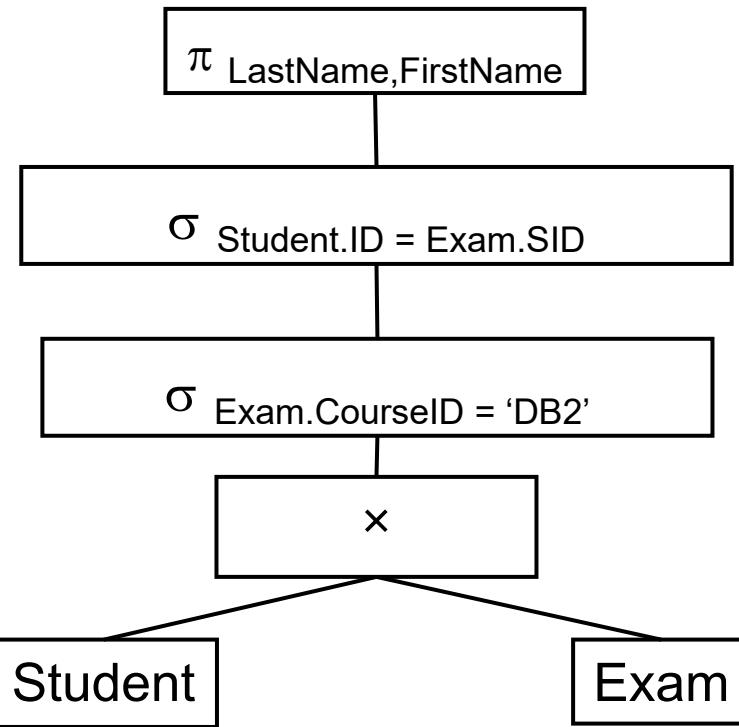
WHERE

Student.ID = Exam.SID

AND

Exam.CourseID = "DB2"

Query Tree



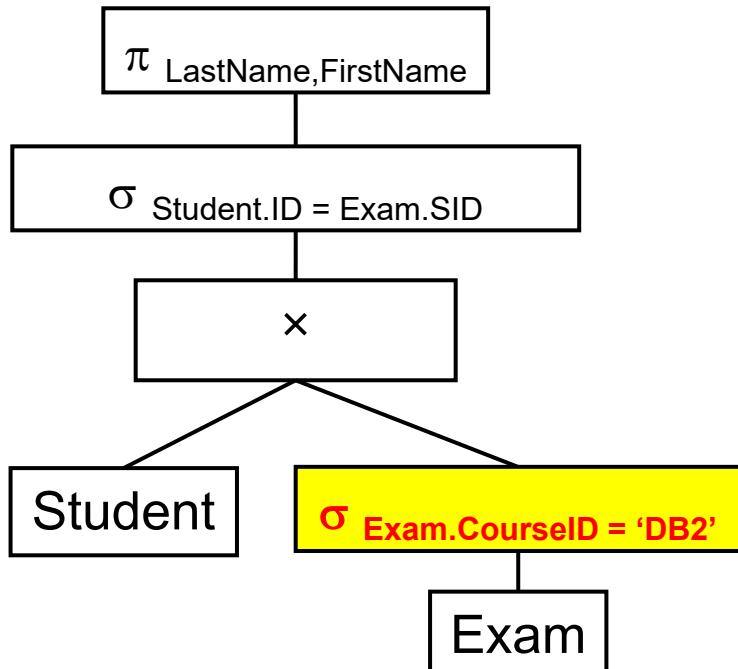
```
SELECT LastName, FirstName  
FROM Student, Exam  
WHERE Student.ID = Exam.SID AND  
Exam.CourseID = "DB2"
```

- Many ways to optimise queries:
 - Changing the query tree to an equivalent but more efficient one
 - Exploiting database statistics
 - Choosing efficient implementations of each operator

Optimisation Example

- Equivalent query:
 - Selecting Exam entries with CourseID = 'DB2'
 - Taking the Cartesian product of the result with Student

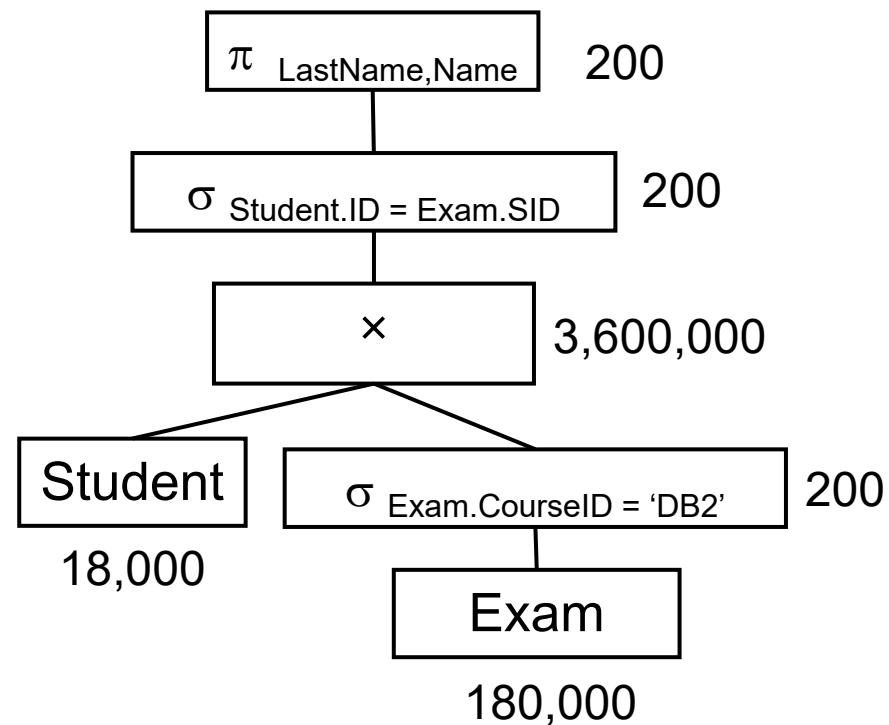
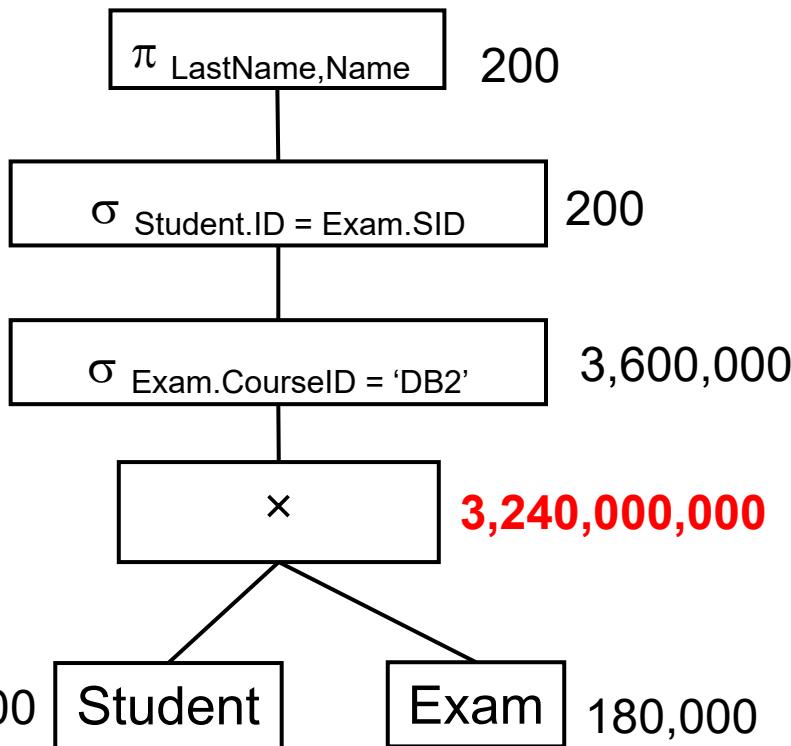
```
SELECT LastName, FirstName  
FROM Student, Exam  
WHERE Student.ID = Exam.SID AND  
Exam.CourseID = "DB2"
```



Optimisation Example

- Consider the following statistics
 - The university has 18,000 students
 - Each student is enrolled in ~10 exams
 - Only 200 take DB2

```
SELECT LastName, FirstName
FROM Student, Exam
WHERE Student.ID = Exam.SID
AND Exam.CourseID = "DB2"
```



Relation profiles

- Profiles are stored in the **data dictionary** and contain quantitative information about tables:
 - the cardinality (number of tuples) of each table T
 - Select count (*) from T may be executed directly on the dictionary without accessing the table data
 - the dimension in bytes of each attribute A_j in T
 - the number of distinct values of each attribute A_j in T : $\text{val}(A_j)$
 - the minimum and maximum values of each attribute A_j in T
 - Queries with WHERE conditions using $<$ and $>$ may be executed directly on the dictionary
- Periodically calculated by activating appropriate system primitives (for example, the **update statistics** command)
- Used in cost-based optimization for estimating the size of the intermediate results produced by the query execution plan

Data profiles and selectivity of predicates

- **Selectivity** = probability that any row will satisfy a predicate
 - If $\text{val}(A) = N$ (attribute A has N values) and
 - the values are homogeneously distributed over the tuples, then
 - the selectivity of a predicate in the form $A=k$ is $1/N$
- Example: $\text{val}(\text{City})=200 \rightarrow \text{selectivity for City} = 1/200 = 0.5\%$
 - 18K students \rightarrow 90 students per city (on average)
- If no data on distributions are available, we will always assume homogeneous distributions!

Optimizations

Operations

- Selection
- Projection
- Sort
- Join
- Grouping

Access methods

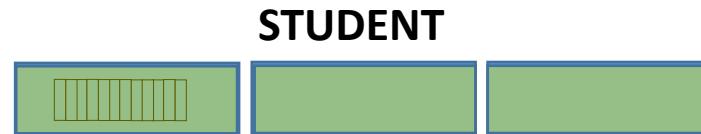
- Sequential
- Hash-based indexes
- Tree-based indexes

Running example

- Block size 8KB (8192 bytes)
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
 - 150K tuples
 - Each tuple is 95 bytes long (4+20+20+20+20+10+1)
 - $t_{\text{STUDENT}} = 8192/95 = 87$ tuples per block (block factor)
 - $b_{\text{STUDENT}} = 150000/87 = 1.7k$ blocks
- EXAM (SID, CourseID, Date, Grade)
 - 1.8 M tuples
 - Each tuple is 24 bytes long (4+4+12+4)
 - $t_{\text{EXAM}} = 340$ tuples per block (block factor)
 - $b_{\text{EXAM}} = 5.3k$ blocks

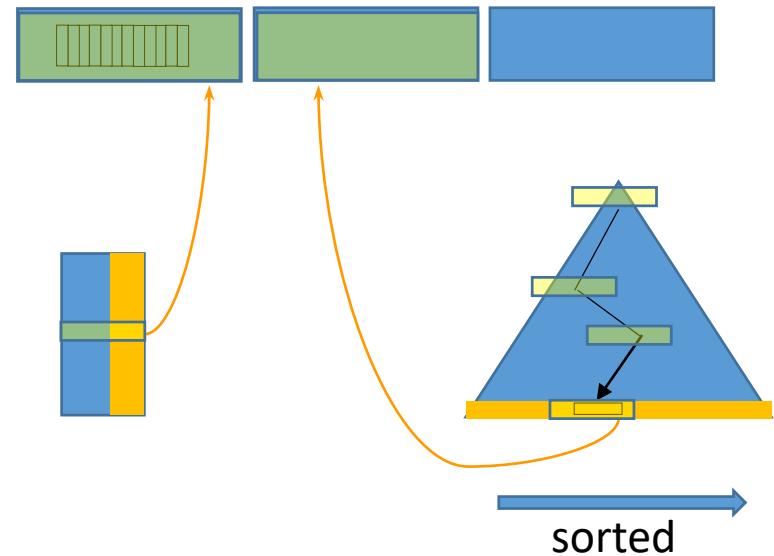
Sequential scan

- Performs a sequential access to all the tuples of a table or of an intermediate result, at the same time executing various operations, such as:
 - Projection to a set of attributes
 - Selection on a simple predicate (of type: $A_i = v$)



Hash and tree based access

- Hash **indexes** support:
 - equality predicates
- Tree-based **indexes** support:
 - selection or join criteria
 - ORDER BY
 - GROUP BY
 - other operations involving sorting (such as UNION or DISTINCT)



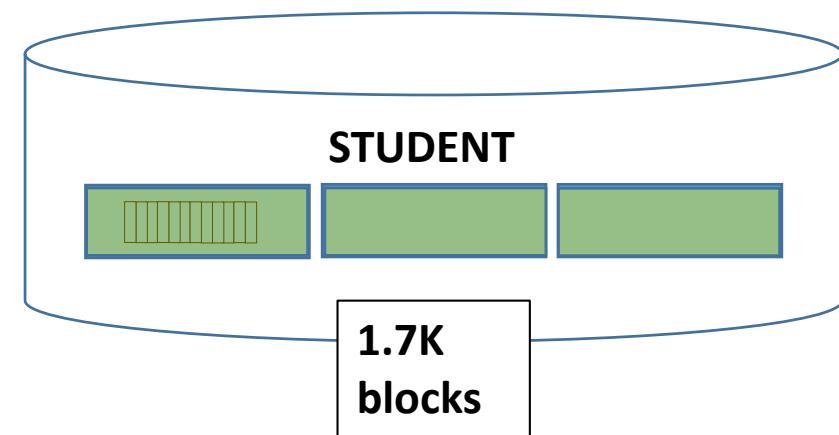
Sequential scan: order key matters

- Example – suppose STUDENT **sequentially ordered by ID**:

```
SELECT *
FROM STUDENT
WHERE City = "Milan"
```

- Cost: **1.7K** I/O accesses

```
SELECT *
FROM STUDENT
WHERE ID < 500
```



- Cost: **< 1.7K** I/O accesses because one can stop when ID=500 (order is by ID)

Cost of lookups: equality ($A_i = v$)

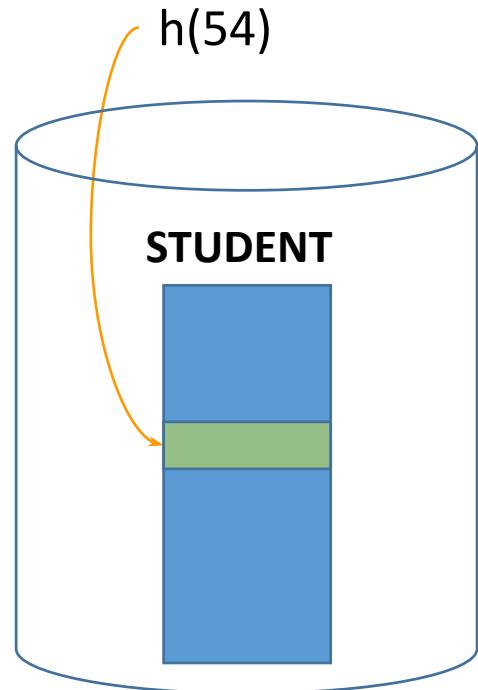
- Sequential structures with no index
 - Lookups are not supported (cost: a full scan)
 - Sequentially-ordered structures may have reduced cost
- Hash/Tree structures
 - Supported if A_i is the search key attribute of the structure
 - The cost depends on
 - the storage type (primary/secondary)
 - the search key type (unique/non-unique)

Equality lookup on a **primary** hash

```
SELECT *
FROM STUDENT
WHERE ID='54'
```

- Query predicate on search key
- Cost:
 - no overflow chains
 - cost = 1
 - With overflow chain with cost 0.3
 - cost = 1.3

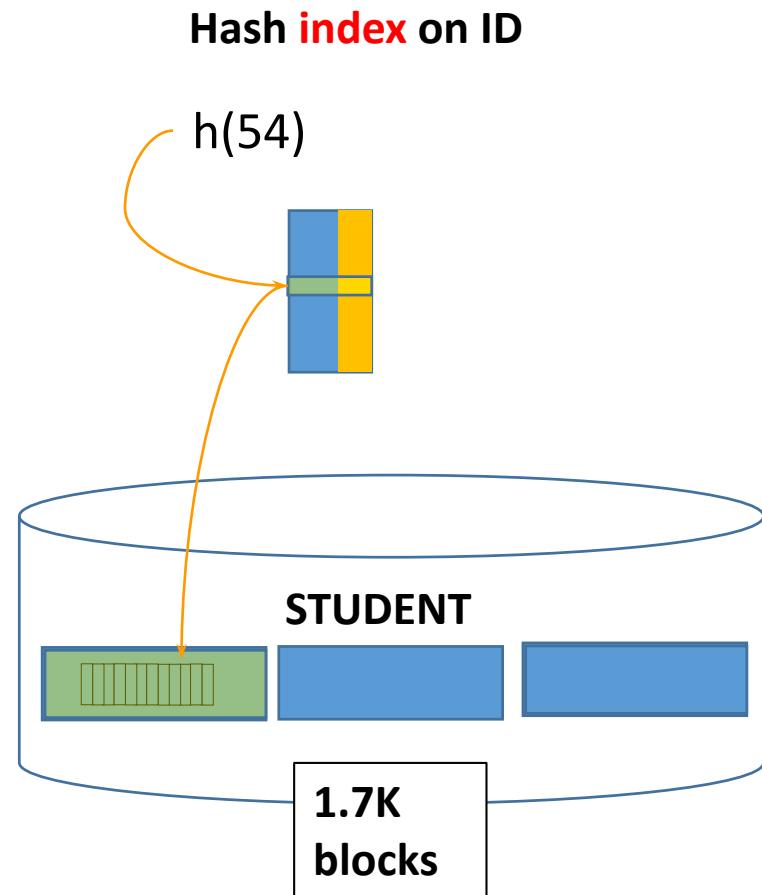
Hash **table** on ID



Equality lookup on a **secondary** hash (unique search key)

```
SELECT *
FROM STUDENT
WHERE ID='54'
```

- Query predicate on search key
- Cost:
 - no overflow chains
 - cost = $1 + 1$
 - With overflow chain with cost 0.3
 - cost = $1.3 + 1$

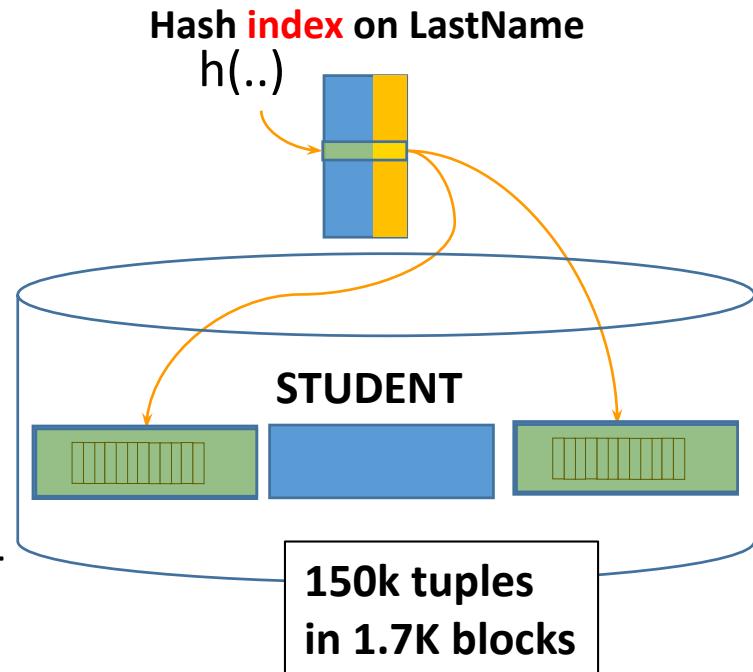


Equality lookup on a **secondary** hash (non unique search key)

```
SELECT *
FROM STUDENT
WHERE LastName="Rossi"
```

- Cost with/without overflow chains:
 - 1 [+ the size of overflow chain (e.g., 0.3)]
 - + cost of accessing blocks in primary storage
- Suppose val(LastName)=75K
 - STUDENT contains 150K tuples → on average each lastname appears twice
 - On average 2 blocks per lastname
- TOTAL COST (with overflow chain) =
 - 1.3 (cost to access the hash index)
+ 2 (cost to access 2 tuples of STUDENT needed for the SELECT *) = 3.3

Note: you cannot assume that the «Rossi» tuples are in the same block of **primary** storage



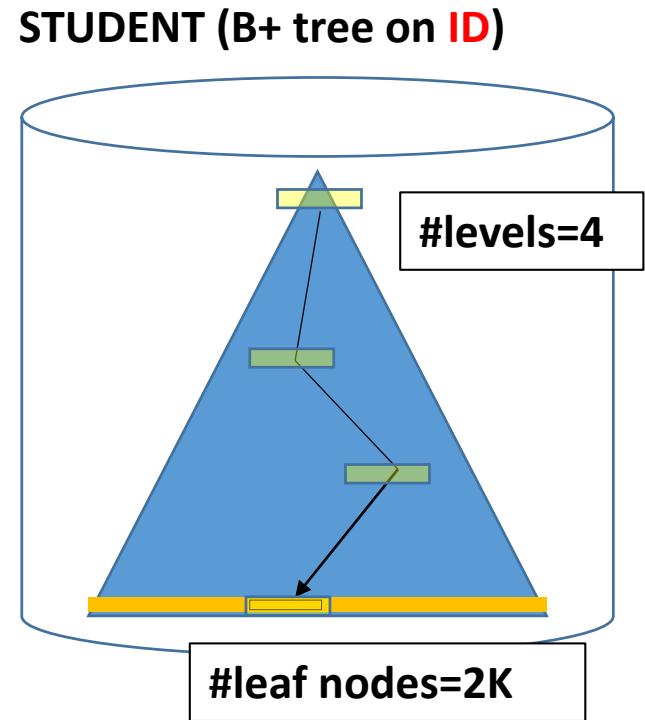
Equality lookup on a **primary** B+

- Query predicate **on unique** search key

- Only 1 tuple per key value

```
SELECT *
FROM STUDENT
WHERE ID=54
```

- Cost:
 - 3 intermediate levels + 1 leaf node = **4**
- Remember: **primary** B+ leaves contain the whole tuples (seldom used..)

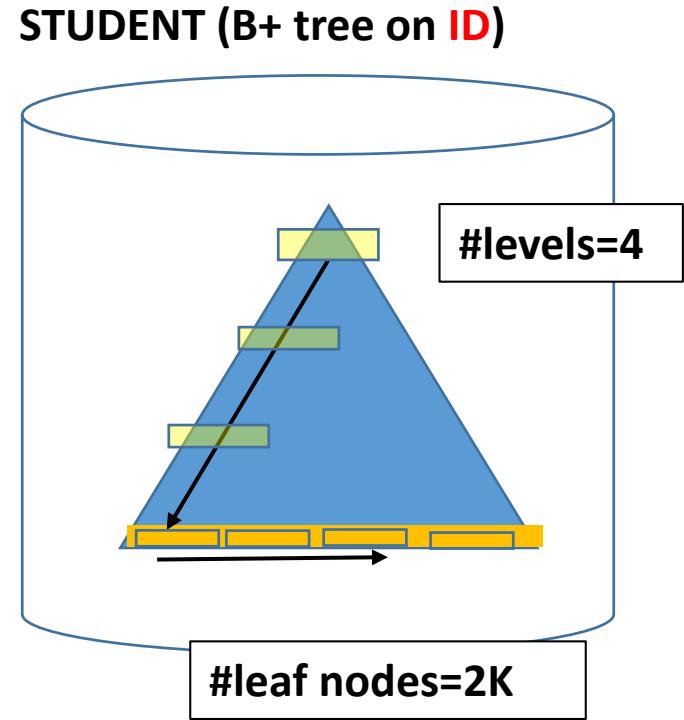


Equality lookup on a **primary** B+

- Query predicate **NOT** on search key

```
SELECT *  
FROM STUDENT  
WHERE city="Milan"
```

- All the tuples are in the leaf nodes, we need to reach & scan all of them
- Cost:
 - 3 intermediate levels + 2K leaf nodes



Equality lookup on a **primary** B+

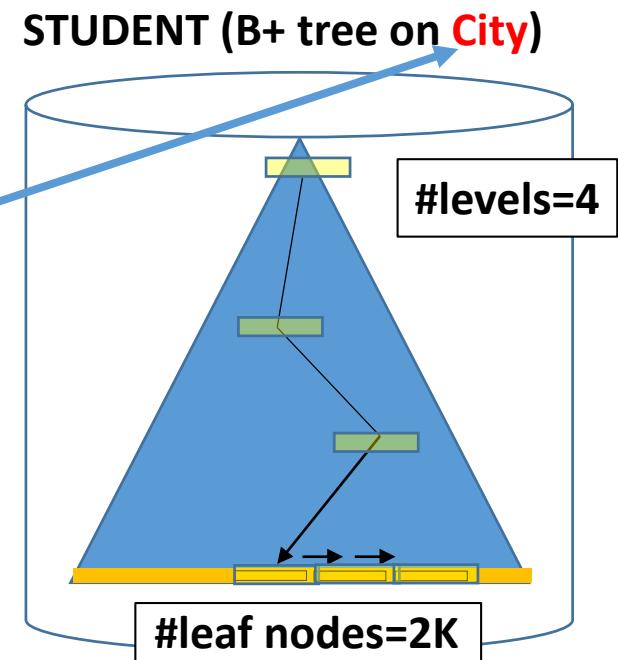
- Query predicate **on non unique** search key

- Multiple tuples per key value

```
SELECT *
FROM STUDENT
WHERE City="Milan"
```

- Cost:

- #levels to access the leaf = 4
 - How many Milan students blocks?



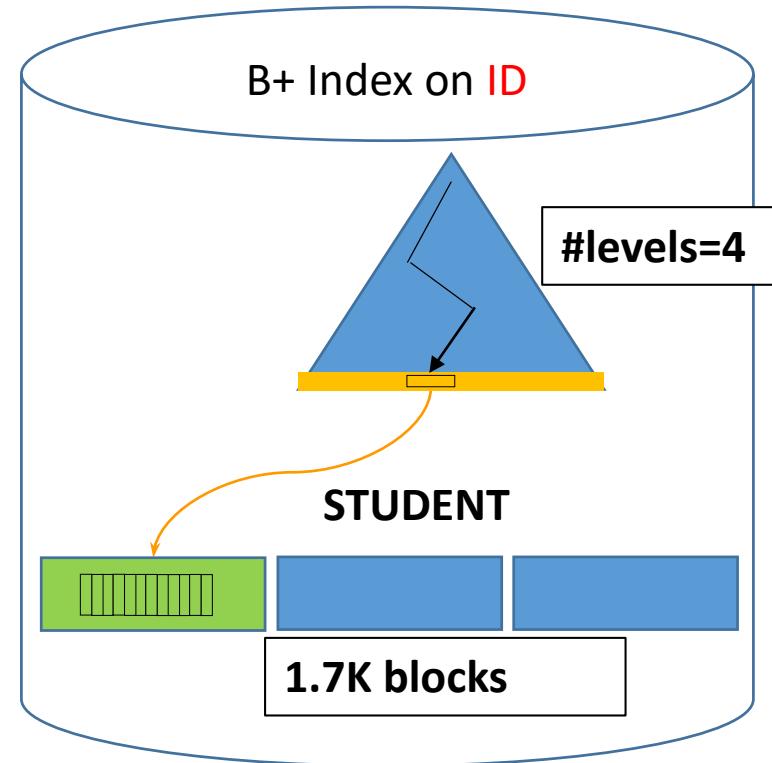
- With statistics: $\text{val}(\text{City}) = 150 \rightarrow 150\text{K} / 150 = 1\text{K tuples/City}$
 - #tuples in 1 leaf node: $150\text{K tuples} / 2\text{K nodes} = 75 \text{ tuples/node}$
 - #blocks with Milan tuples:
 - $1\text{K Milan tuples} / 75 \text{ tuples/node} = 13.33 \rightarrow 14 \text{ leaf nodes sequentially chained}$
 - Total cost
 - = 3 intermediate levels + 14 leaf blocks = 17

Equality lookup on **secondary** B+

- Query predicate **on unique** search key of STUDENT
 - Only 1 pointer per key value

```
SELECT *
FROM STUDENT
WHERE ID="54"
```

- Cost:
 - 3 intermediate levels + 1 leaf node + 1 data block = 5

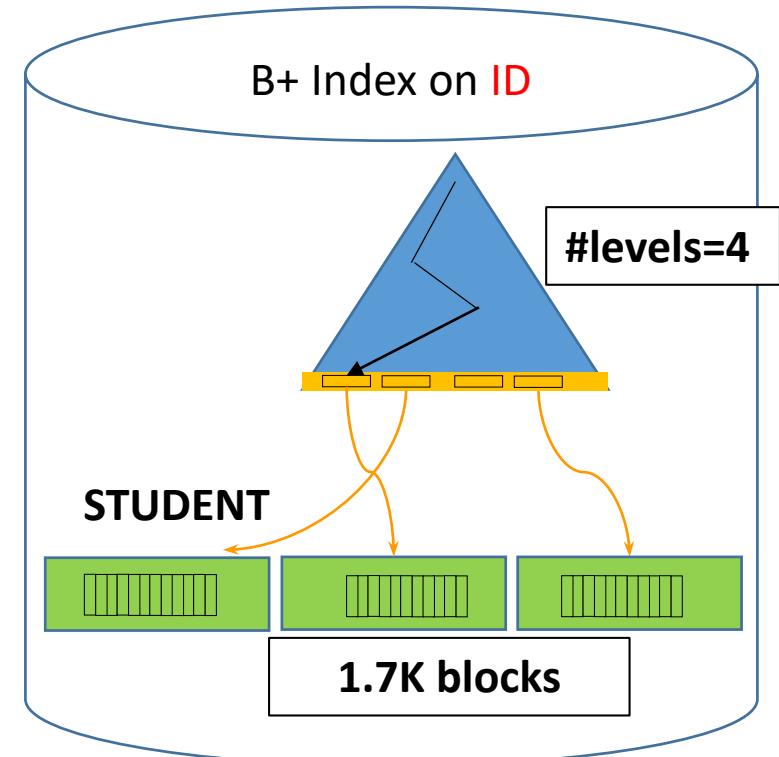


Equality lookup on **secondary** B+

- Query predicate **not on** search key of STUDENT

```
SELECT *
FROM STUDENT
WHERE city="Milan"
```

- Cost:
 - Index is not useful for this query (full scan costs 1.7k)



Equality lookup on **secondary** B+

- Query predicate **on non unique** search key of STUDENT

- Multiple pointers per key value

```
SELECT *
```

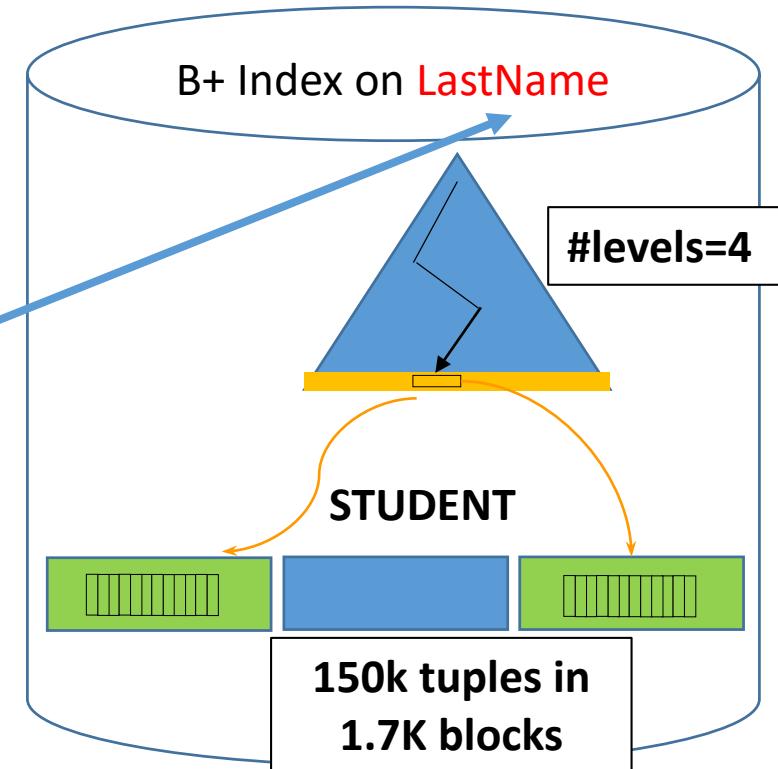
```
FROM STUDENT
```

```
WHERE LastName="Rossi"
```

- Suppose $\text{Val}(\text{LastName}) = 75\text{K}$
→ **2 tuples / Lastname**

- Cost:

- #of levels of the B+tree: **3 + 1**
(2 pointers fit in 1 leaf node)
 - #of blocks to be accessed in STUDENT following the B+ tree pointers in the leaf: **2**
 - Total cost = $3 + 1 + 2 = 6$



We assume that we reload a block for tuple T_i even if we have loaded it already for a tuple T_j (no caching)

Equality lookup on a **secondary** B+

- Query predicate **on non unique** search key of EXAM

- Multiple pointers per key value

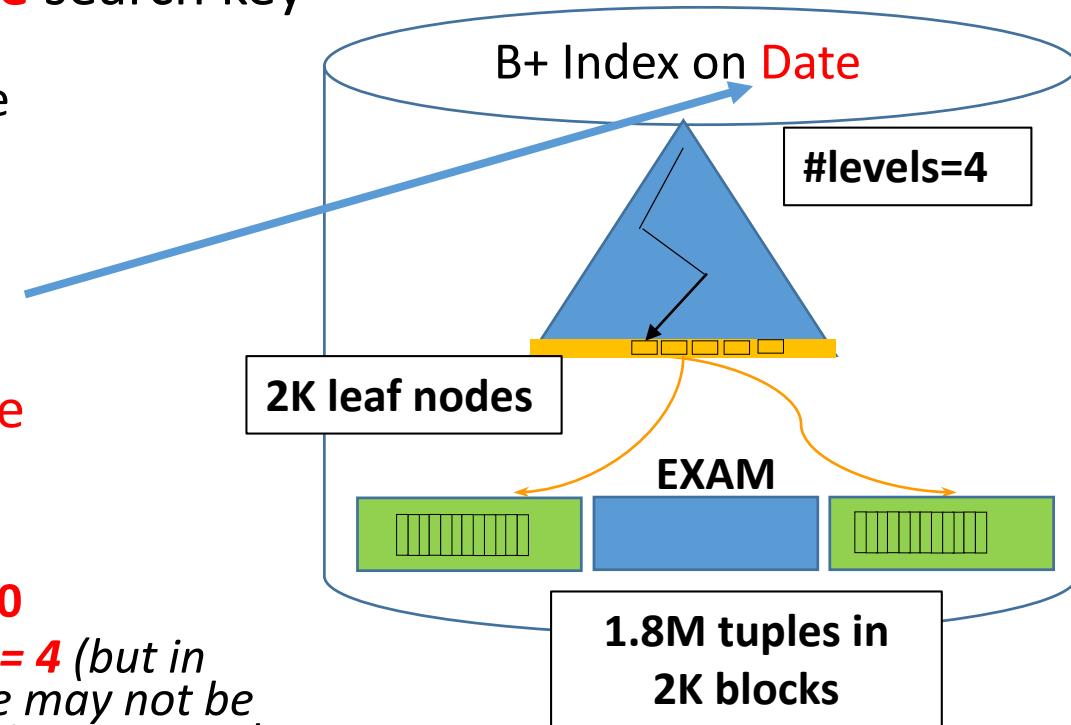
```
SELECT *
FROM EXAM
WHERE Date="10/6/2019"
```

- Suppose $\text{val}(\text{Date}) = 500$
 $\rightarrow 1.8\text{M} / 500 = 3.6\text{K tuples/date}$

- Cost factors =

- #of intermediate levels **3**
- pointers per leaf **$1.8\text{M} / 2\text{K} = 900$**
- leaf nodes per date: **$3.6\text{K} / 900 = 4$** (*but in most of the cases, the given date may not be the left-most key in the block \rightarrow in average the 900 pointers will be in 5 blocks*)
- blocks to access in EXAM (1 per pointer) = **3.6K**

- Total cost = $3 + 5 + 3.6\text{K} = \sim 3.6\text{K}$**



We assume that we reload a block for tuple T_i even if we have loaded it already for a tuple T_j (no caching)

Interval lookups $A_i < v$, $v_1 \leq A_i \leq v_2$

- Sequential structures (primary)
 - Lookups are **not supported** (cost: a full scan)
 - Sequentially-ordered structures may have reduced cost
- Hash structures (primary/secondary)
 - Lookups based on intervals are **not supported**
- Tree structures (primary/secondary)
 - Supported **if A_i is the search key**
 - The cost depends on
 - the storage type (primary/secondary) – see next slides
 - the index key type (unique/non-unique)

Interval lookup on primary B+

- We consider a lookup for $v_1 \leq A_i \leq v_2$ as the general case
 - If $A_i < v$ or $v < A_i$ we just assume that the other edge of the interval is the first/last value in the structure
- The root is read first
 - then, a node per intermediate level is read, until...
- ...the first leaf node is reached, that stores tuples with $A_i = v_1$
 - If the searched tuples are all stored in that leaf block, stop
 - Else, continue in the leaf blocks chain until v_2 is reached
- Cost: **1** block per intermediate level + **as many leaf blocks as necessary** to read all the tuples in the interval (estimated with statistics)

Interval lookup on secondary B+

- We still consider a lookup for $v_1 \leq A_i \leq v_2$ as the general case
- The root is read first
 - then, a node per intermediate level is read, until...
- ...the first leaf node is reached, that stores the pointers pointing to the blocks containing the tuples with $A_i = v_1$
 - If all the pointers (up to v_2) are in that leaf block, stop
 - Else, continue in the leaf blocks chain until v_2 is reached
- Cost: **1 block per intermediate level + as many leaf blocks as necessary to read all pointers in the interval + 1 block per each such pointer** (to retrieve the tuples)

Conjunction / disjunction

- **Predicates in conjunction**

```
SELECT * FROM STUDENT  
WHERE City="Milan" AND Birthdate="12/10/2000"
```

- If supported by indexes, the DBMS chooses the **most selective** supported predicate for the data access, and evaluates the other predicates in main memory

- **Predicates in disjunction:**

```
SELECT * FROM STUDENT  
WHERE City="Milan" OR Birthdate="12/10/2000"
```

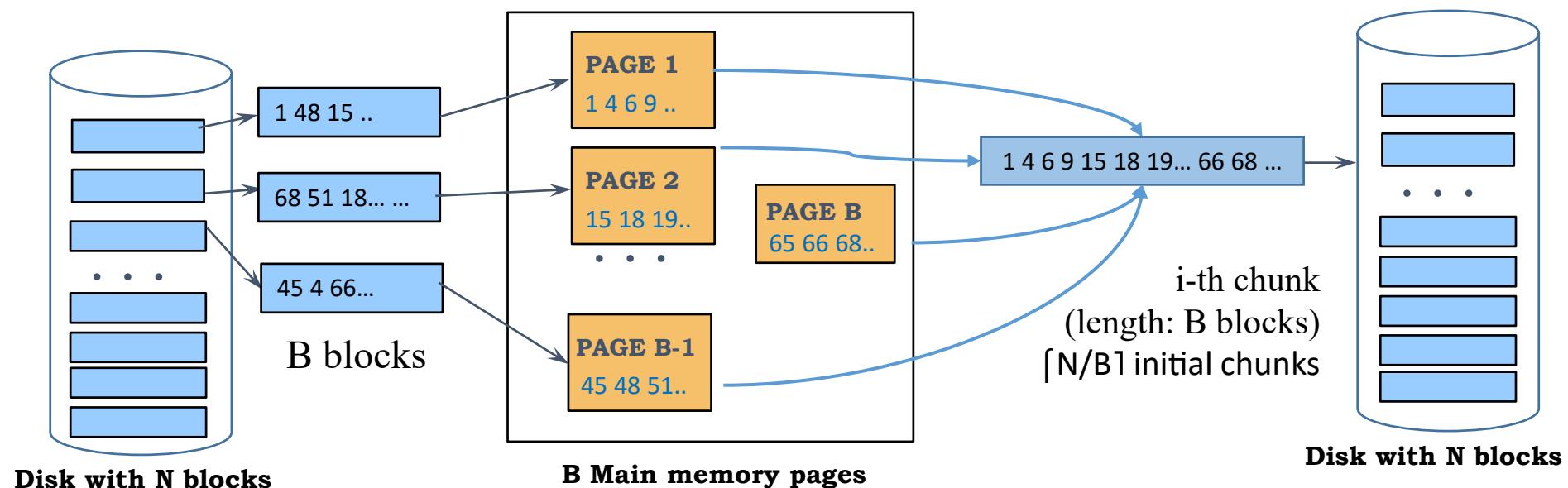
- if any of the predicates is not supported by indexes, then a scan is needed
- if **all** are supported, indexes can be used to evaluate all predicates and then duplicate elimination is normally required

Sort

- This operation is used for ordering the data according to the value of one or more attributes. We distinguish:
 - Sort in main memory, typically performed by means of ad-hoc algorithms (merge-sort, quicksort, ...)
 - Sort of large files, performed with different algorithms such as
 - **External merge-sort:**
 - Data do not fit into the main memory
 - Procedure:
 1. Load from disk as much data as fit in main memory and sort them, store such sorted chunks back to disk
 2. Then, merge sorted chunks parts using at least three pages: two for progressively loading data from two sorted chunks and one as an output buffer to store the merge-sorted data. Save the merged-sort chunks back to disk
 3. Repeat step 2 until all data are sorted

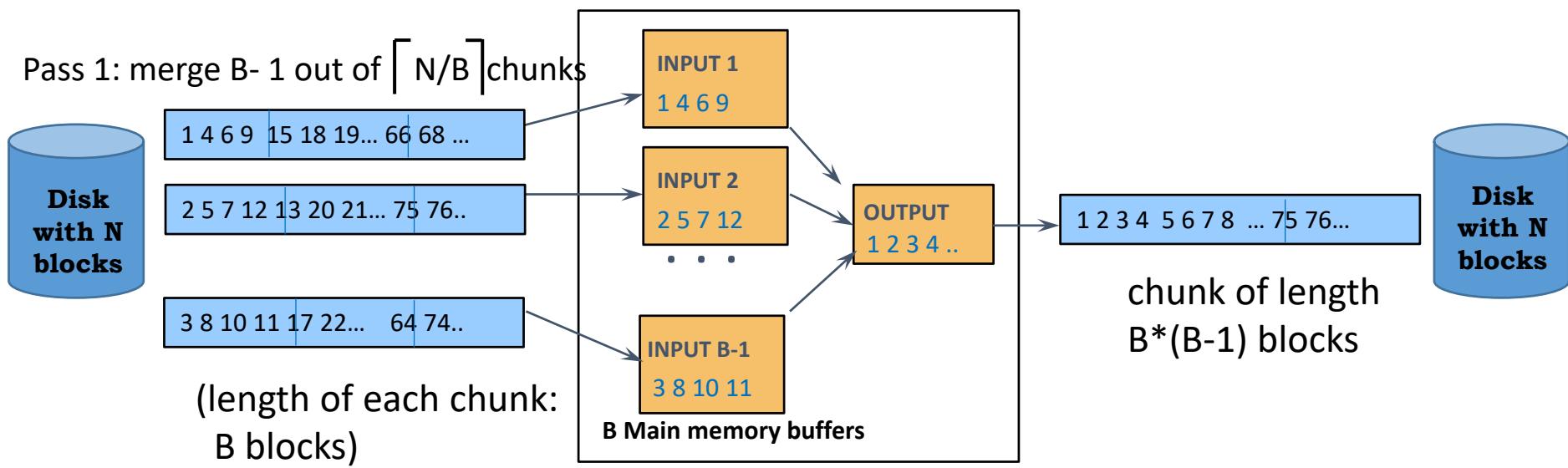
External Merge Sort

- To sort a file stored in N blocks using B buffer pages:
 - Pass 0:
 - read B blocks at a time into main memory and sort them;
 - write sorted data to a chunk file (also called “run”)
 - \rightarrow total chunks = $\lceil N/B \rceil$



External Merge Sort

- Pass 1, 2, 3, 4, ...: Use $B-1$ blocks for input chunks, 1 block for OUTPUT
- Repeat
 - Read first block from each chunk into a buffer page
 - Select the first record (in sort order) among all buffer pages and write it to the output buffer; if the output buffer is full, write it to disk
 - Delete the record from its input buffer page. If the buffer page is empty, read next block of the chunk into the buffer
- Until all input buffer pages are empty

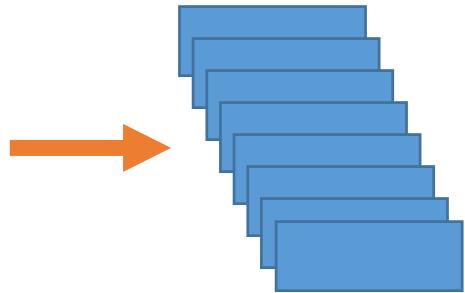


External Merge Sort

- In each pass, $B-1$ chunks are merged into a larger one
- Repeated passes are performed until all chunks have been merged into one
- A pass reduces the number of chunks by a factor of $B-1$ and creates chunks longer by the same factor
- E.g., $B=5$, $N=40 \rightarrow$ initial chunks = $40/5 = 8$ (having length 5 pages)
 - 40 read operations + 40 write operations
- In the next pass: with 4 input pages + 1 output page
 - First load 4 chunks and create a new chunk having length $5*4=20$ pages
 - Load the next 4 chunks and create a new chunk having length 20 pages
 - $8/4 = 2$ sorted chunks of 20 pages each $\rightarrow 40 + 40$ I/O operations
- In the next pass load the 2 final chunks into the final sorted chunk of length $20*2$
 - $2/4 = 1$ sorted chunks of 40 pages $\rightarrow 40 + 40$ I/O operations
 - Total passes = 3

$B=5, N=40$

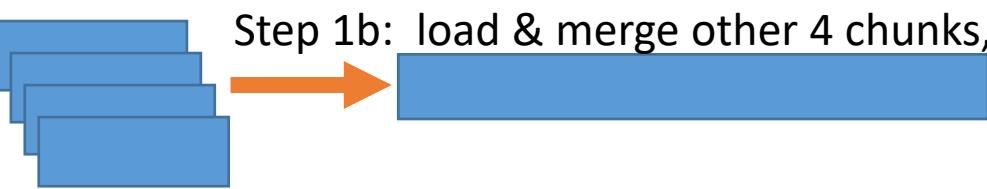
Step 0: create 8 ($40/5$) sorted chunks of 5 blocks



Step 1a: load & merge first 4 chunks, create one chunk of $5 \times 4 = 20$ blocks



Step 1b: load & merge other 4 chunks, create 2° chunk of 20 blocks



Step 2: load & merge 2 chunks of 20 blocks create one chunk of 40 blocks



External sorting

- Number of passes count =
 - 1 (for initial step) + $\lceil \log_{B-1} \lceil N/B \rceil \rceil$ (for iterative merge of chunks)
 - In the example: $1 + \log_4 8 = 1 + 2$
- Cost = 2 (1 for reading + 1 for writing) * N * (# of passes)

# of blocks	N	# of buffers in main memory					
		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

A blue arrow points from the "# of passes" label to the last column of the table.

Join Methods

- Joins are the most frequent (and costly) operations in DBMSs
- There are several join strategies, among which:
 - nested-loop, merge-scan and hashed
 - These three join methods are based on scanning, ordering and hashing
- The “best” strategy is chosen based on various aspects...

Running example

- Block size 8KB (8192 bytes)
- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
 - 150K tuples
 - Each tuple is 95 bytes long (4+20+20+20+20+10+1)
 - $t_{\text{STUDENT}} = 8192/95 = 87$ tuples per block (block factor)
 - $b_{\text{STUDENT}} = 150000/87 = 1.7k$ blocks
- EXAM (SID, CourseID, Date, Grade)
 - 1.8 M tuples
 - Each tuple is 24 bytes long (4+4+12+4)
 - $t_{\text{EXAM}} = 340$ tuples per block (block factor)
 - $b_{\text{EXAM}} = 5.3k$ blocks

Equality Joins With One Join Column

```
SELECT  *
FROM  Student, Exam
WHERE ID=SID
```

- In algebra: $\text{Student} \bowtie_{ID=SID} \text{Exam}$
 - Common! Must be carefully optimized
 - $\text{Student} \times \text{Exam}$ is large \rightarrow $\text{Student} \times \text{Exam}$ followed by a selection is inefficient

Nested-Loop join

- Scan the external table and for each block scan the internal table

External table T_{Ext} (b_{Ext} blocks, t_{Ext} tuples)

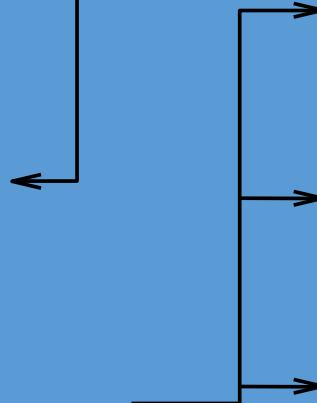
K1	A	B	C
a			

Internal table T_{Int} (b_{Int} blocks, t_{Int} tuples)

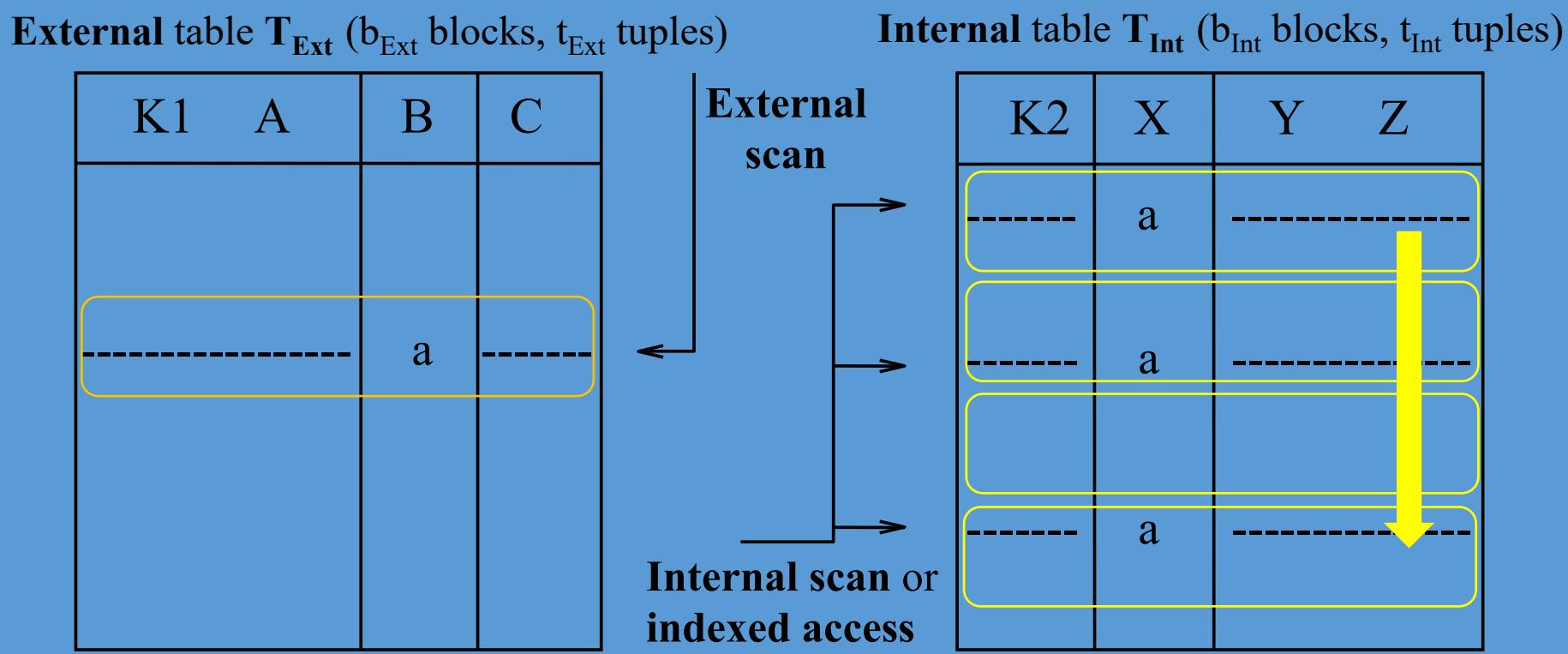
K2	X	Y	Z
a			
a			
a			

External
scan

Internal scan or
indexed access



Nested-Loop join



Cost of simple nested-loop joins

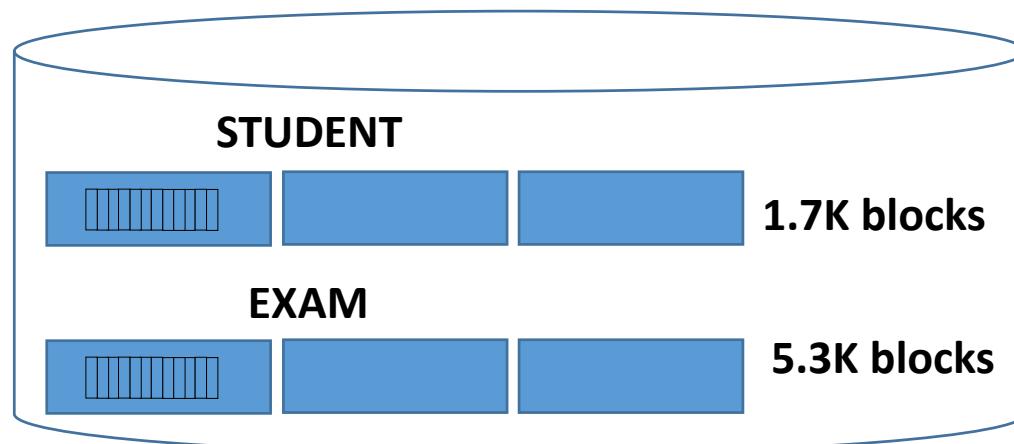
- A nested loop join compares the tuples of a block of table T_{Ext} with all the tuples of all the blocks of T_{Int} before moving to the next block of T_{Ext}
 - We always assume that the buffer does not have enough available free pages to host more than a few blocks
 - The cost is **quadratic** in the size of the tables: scan the external table and then scan the internal table once for each block of the external one
 - $\text{Cost} = b_{Ext} + b_{Ext} \times b_{Int} = b_{Ext} \times (1 + b_{Int}) = \sim b_{Ext} \times b_{Int}$
 - **Cached nested loop:** if one of the tables is small enough to fit in the buffer, then it is chosen as internal table and scanned **only once**.
 - $\text{Cost} = b_{Ext} + b_{Int}$

A simple Nested-Loop join

- STUDENT (Id, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

```
SELECT STUDENT.*  
      FROM STUDENT JOIN EXAM ON ID=SID
```

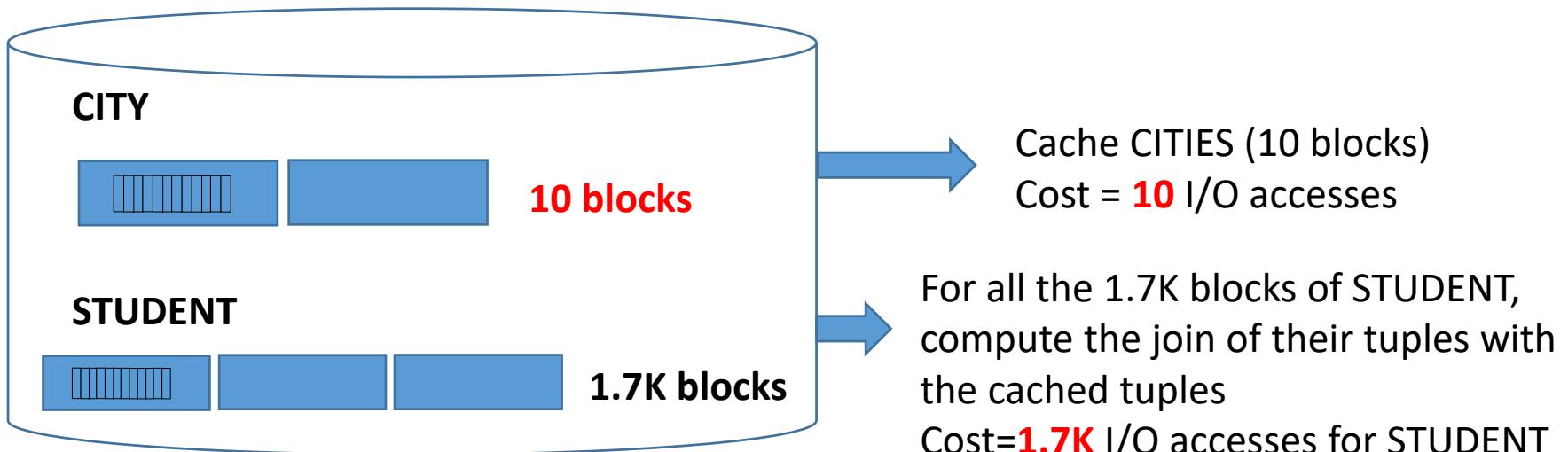
- Student external table, Exam internal table
 - Cost = $1.7K + 1.7K * 5.3K = \sim 9M$ I/O accesses
- Exam external table, Student internal table
 - Cost = $5.3K + 5.3K * 1.7K = \sim 9M$ I/O accesses



Nested-Loop join with cache

- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- CITIES (City, Region, Description)
 - $b_{CITIES} = 10$ blocks, with block size 8KB $\rightarrow 80$ KB \rightarrow cacheable

SELECT STUDENT.* FROM STUDENT NATURAL JOIN City



$$\text{Total cost} = 10 + 1.7K \approx \mathbf{1,7K}$$

Filtering by condition: option 1

- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

SELECT STUDENT.*

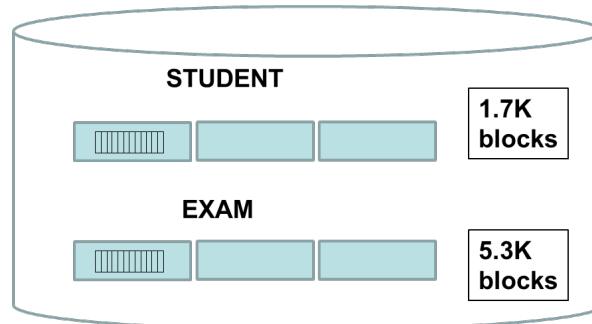
FROM STUDENT JOIN EXAM ON ID=SID

WHERE City='Milan' AND Grade='30'

val(City) = 150
val(Grade) = 17

Option 1: Scan Student, evaluate City='Milan' + Lookup on Exam

- Read all the STUDENT blocks: **1.7K**
- **Filter** students from Milan → 150K tuples / 150 cities = **1k**
- For 1k students do the join with the tuples of all the 5.3K blocks in EXAM → this requires 1k scans



Total cost:
1,7K + 1K*5.3K
= ~ **5.3M** I/O accesses

Filtering by condition: option 2

- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

SELECT STUDENT.*

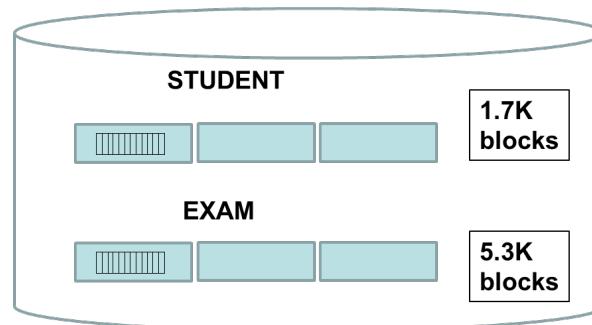
FROM STUDENT JOIN EXAM ON ID=SID

WHERE City='Milan' AND Grade='30'

val(City) = 150
val(Grade) = 17

Option 2: Scan Exam, evaluate Grade='30' + Lookup on Student

- Read all the EXAM blocks: **5.3K**
- **Filter** Exams with Grade='30' → 1.8M/17 different grades = **106K**
- For 106K exams do the join with 1.7K blocks in STUDENT



Total cost:
 $5,3K + 106K * 1.7K$
 $= \sim 180M$ I/O accesses

Scan and lookup (indexed nested loop)

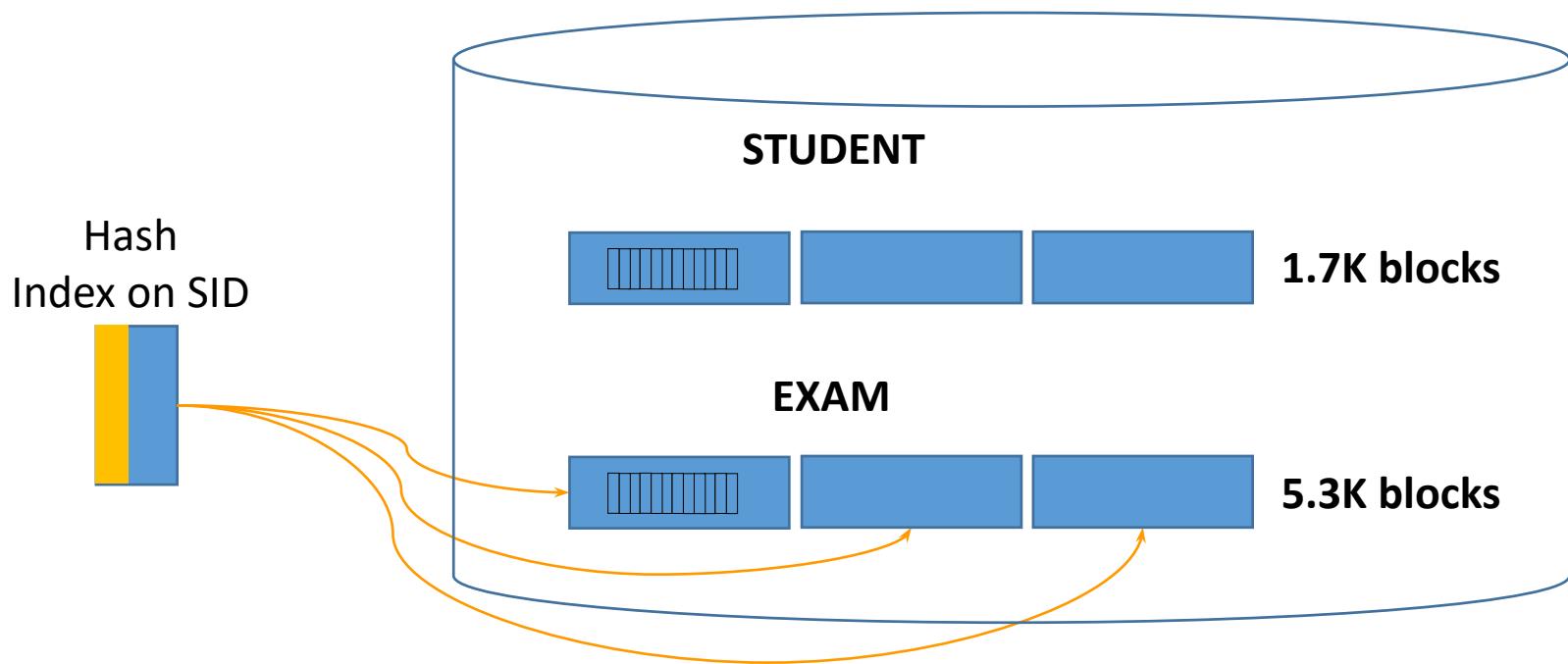
- IF one table supports **indexed access in the form of a lookup based on the join predicate**, then this table can be chosen as **internal**, exploiting the predicate to extract the joining tuples without scanning the entire table
 - If both tables support lookup on the join predicate, the one for which the predicate is more selective is chosen as internal
- Cost:
 - full scan of the blocks of the external table +
 - cost of lookup for each tuple of the external table onto the internal one, to extract the matching tuples
 - $\text{Cost} = b_{\text{Ext}} + t_{\text{Ext}} \times \text{cost_of_one_indexed_access_to_T}_{\text{Int}}$
 - NOTE: as before, if the query has a filtering predicate on the external table then t_{Ext} is the subset of the tuples of the external table that satisfy the predicate (estimated)

Scan & lookup exploiting an index

- STUDENT (ID, Name, LastName, Email, City, Birthdate, Sex)
- EXAM (SID, CourseID, Date, Grade)

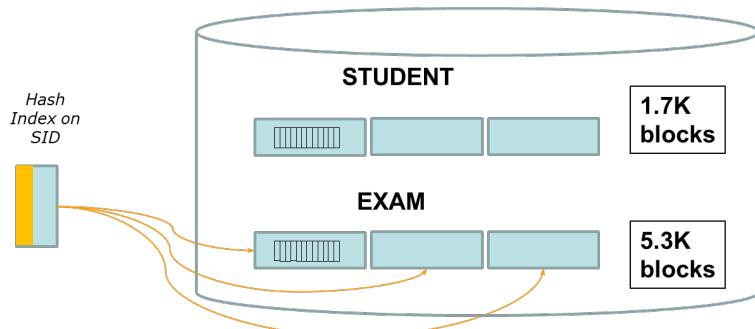
```
SELECT STUDENT.*  
FROM STUDENT JOIN EXAM ON ID=SID  
WHERE City='Milan' AND Grade='30'
```

val(City)=150



Filter + lookup: option 1 revised

- Improve option 1 (STUDENT as external table):
 - Scan Student, evaluate City='Milan' + Lookup on Exam **exploiting the secondary hash index on EXAM**
- Scan all the STUDENT blocks 1.7K
- Filter students from Milan -> 150K tuples / 150 different cities = 1k
- Lookup with the hash index: for 1k students
 - 1 access to the hash index (without overflow chain)
 - How many exams per student? 1.8M exams / 150K students = 12
 - Follow the 12 pointers to retrieve the exams (and the **grades needed to evaluate the where condition**)



Hash index access for Milan students

Student scan

Total cost =

$1.7K + 1K + 1k * 12 =$

$1.7K + 1K * (1 + 12) = 14.7K$

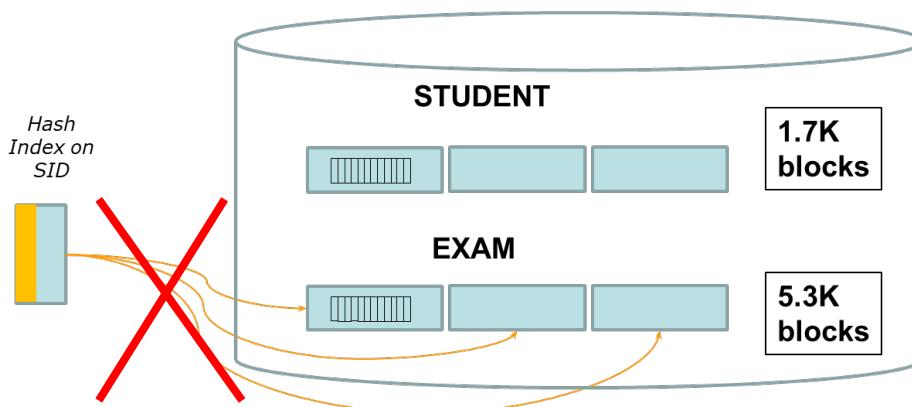
Exam access for Milan students

Existence checking lookup

```
SELECT STUDENT.*  
FROM STUDENT JOIN EXAM ON Id=SID  
WHERE City='Milan' AND Grade='30'
```

val(City)=150

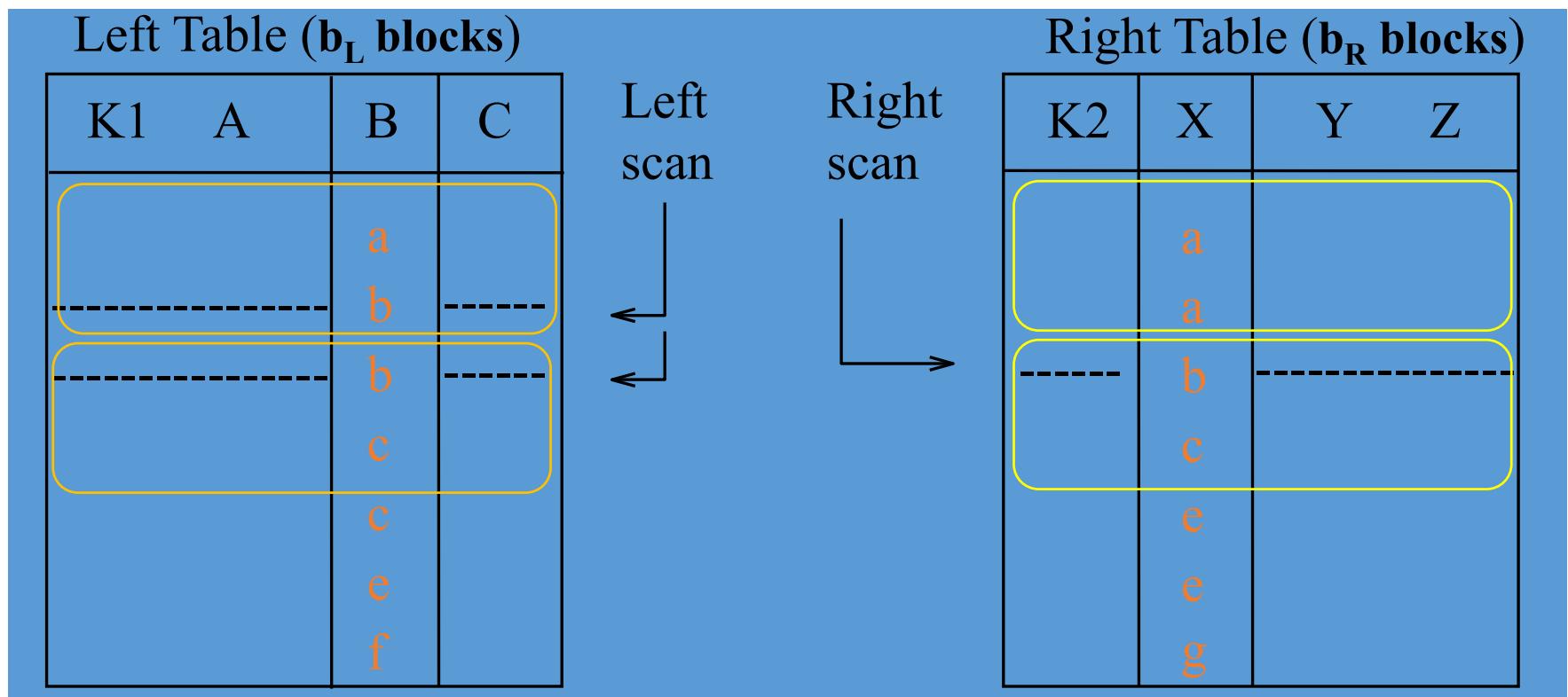
- Read all the STUDENT blocks **1.7K**
- Filter students from Milan → 150K tuples/150 different cities = **1k**
- For 1k students use the hash index: for each student
 - access to the hash index (without overflow chain): **1**
- **No need to access the full EXAM tuples to do the join, only check that the joined tuple pair exists**



Total cost =
 $1.7K + 1K * (1 + 1) = 2.7K$

Merge-scan join

- This join is possible only if both tables are ordered according to the same key attribute, that is also the attribute used in the join predicate

$$L \bowtie_{B=X} R$$


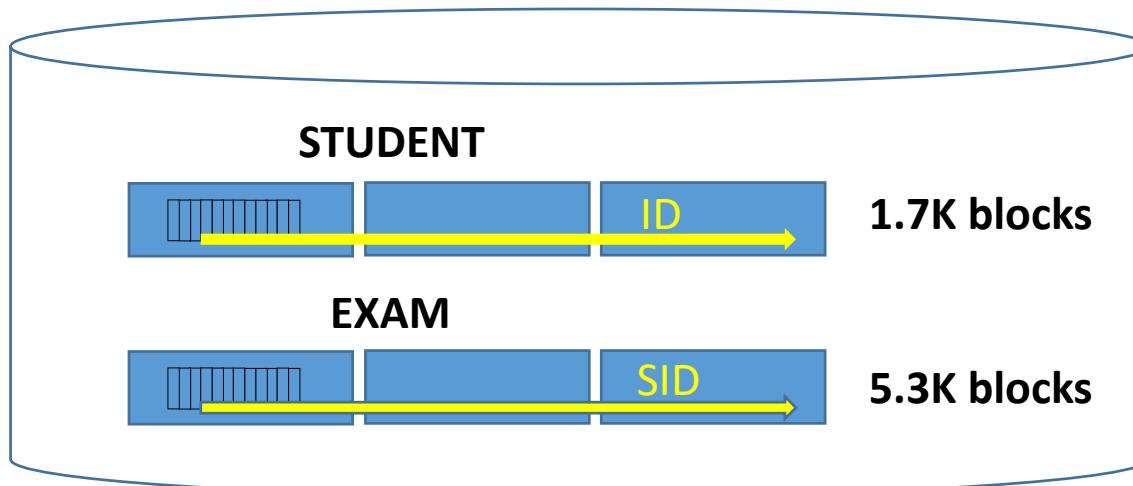
Sort + merge-scan join

- If not sorted, sort L and R **on the join column**
- Scan them to do a “merge”, advancing on the tables with the least value of the join attribute
- Output result tuples $\langle l, r \rangle$
- The cost is **linear in the size of the tables**
 - $C = b_L + b_R$
- The ordered full scan is possible for a table **if the primary storage is sequentially ordered wrt the join attribute or a B+ on the join attribute is defined**
- If a table needs to be sorted, add also the cost for sorting
 - $2 b_L * (\# \text{ of passes})$
 - $2 b_R * (\# \text{ of passes})$
 - # of passes cost = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$, where $N = \# \text{ of blocks}$, $B = \# \text{ of buffer pages}$

Merge scan join on ordered tables

```
SELECT STUDENT.*  
FROM STUDENT JOIN EXAM ON ID=SID
```

- Suppose
 - STUDENT: Sequentially-ordered by ID
 - EXAM: Sequentially-ordered by SID

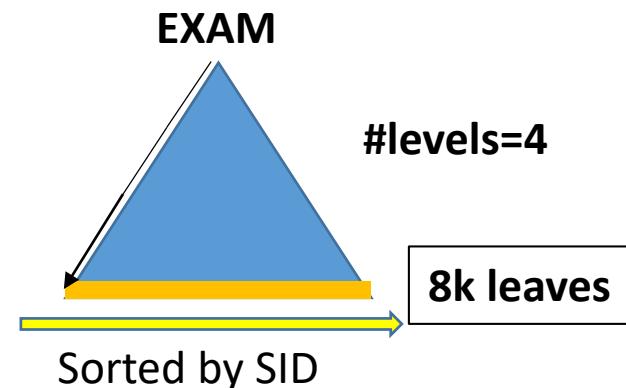
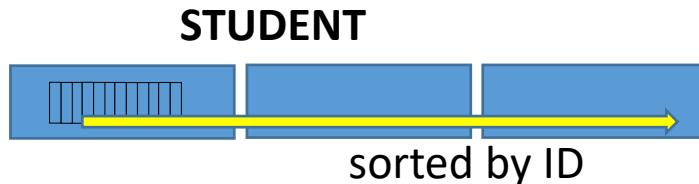


$$C = b_{\text{Student}} + b_{\text{Exam}} = 1.7K + 5.3K = \mathbf{7K} \text{ I/Os}$$

Indexed merge scan join

```
SELECT STUDENT.*  
FROM STUDENT JOIN EXAM ON ID=SID
```

- Suppose
 - STUDENT: Sequentially-ordered by ID
 - EXAM: **primary** storage: B+ on SID (e.g., with 8k leaves)
- $C = b_{\text{Student}} + \#\text{intermediateNodes}_{\text{Exam}} + \#\text{LeafNodes}_{\text{Exam}}$
- $C = 1.7k + 3 + 8k = \sim 9.7k$



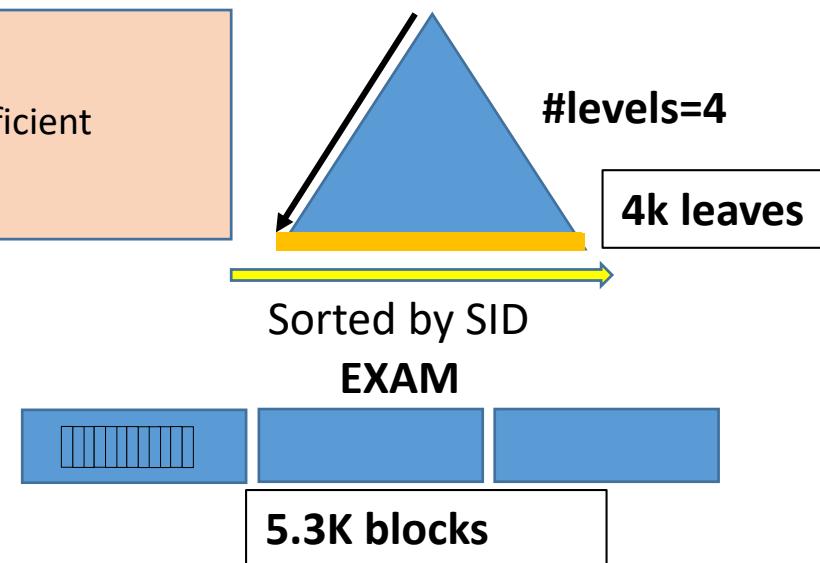
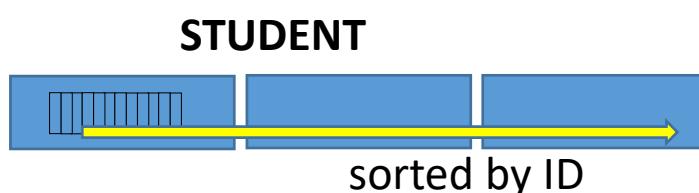
Indexed merge scan join

```
SELECT STUDENT.*  
FROM STUDENT JOIN EXAM ON ID=SID
```

- Suppose
 - STUDENT: Sequentially-ordered by ID
 - EXAM: **secondary** index: B+ on SID
- $C = b_{\text{Student}} + \#\text{intermediateNodes}_{\text{Exam}} + \#\text{LeafNodes}_{\text{Exam}}$
- $C = 1.7k + 3 + 4k = \sim 5.7k$

• NOTE:

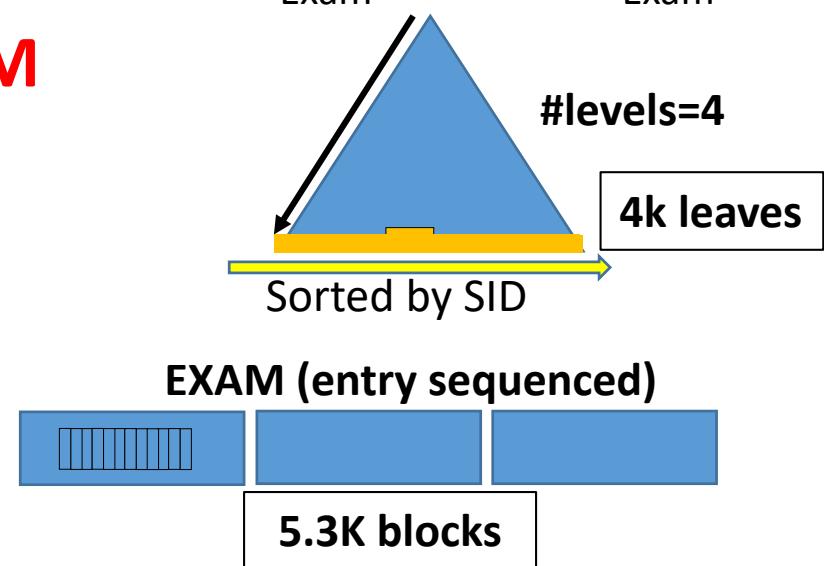
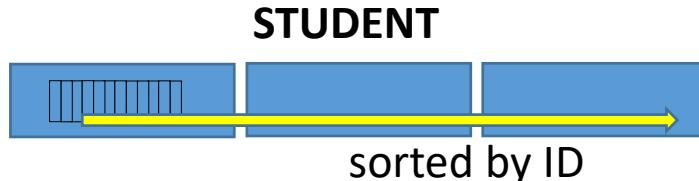
- Exam tuples are not needed, so B+ leaves are sufficient
- One pointer <--> one tuple (SID is the PK)



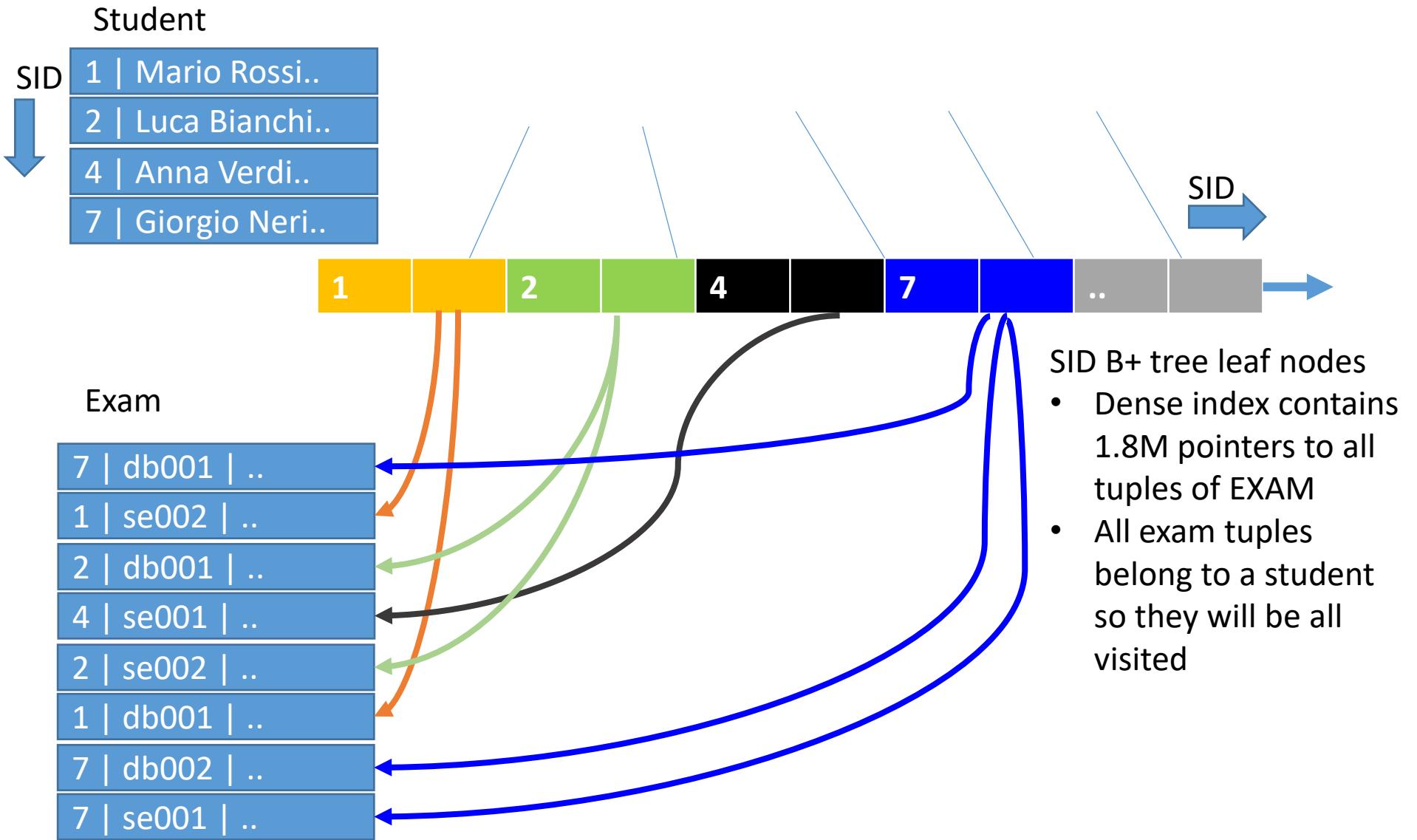
Indexed merge scan join

```
SELECT S.SID  
FROM STUDENT S JOIN EXAM E ON ID=SID  
WHERE GRADE>17 -- exam tuples must be accessed too
```

- Suppose
 - STUDENT: Sequentially-ordered by ID
 - EXAM: **secondary** index: B+ on SID
 - No statistics on grades
- $C = b_{\text{Student}} + \#\text{intermediateNodes} + \#\text{LeafNodes}_{\text{Exam}} + \#\text{Tuples}_{\text{Exam}}$
- $C = 1.7k + 3 + 4k + 1.8M = \sim 1.8M$

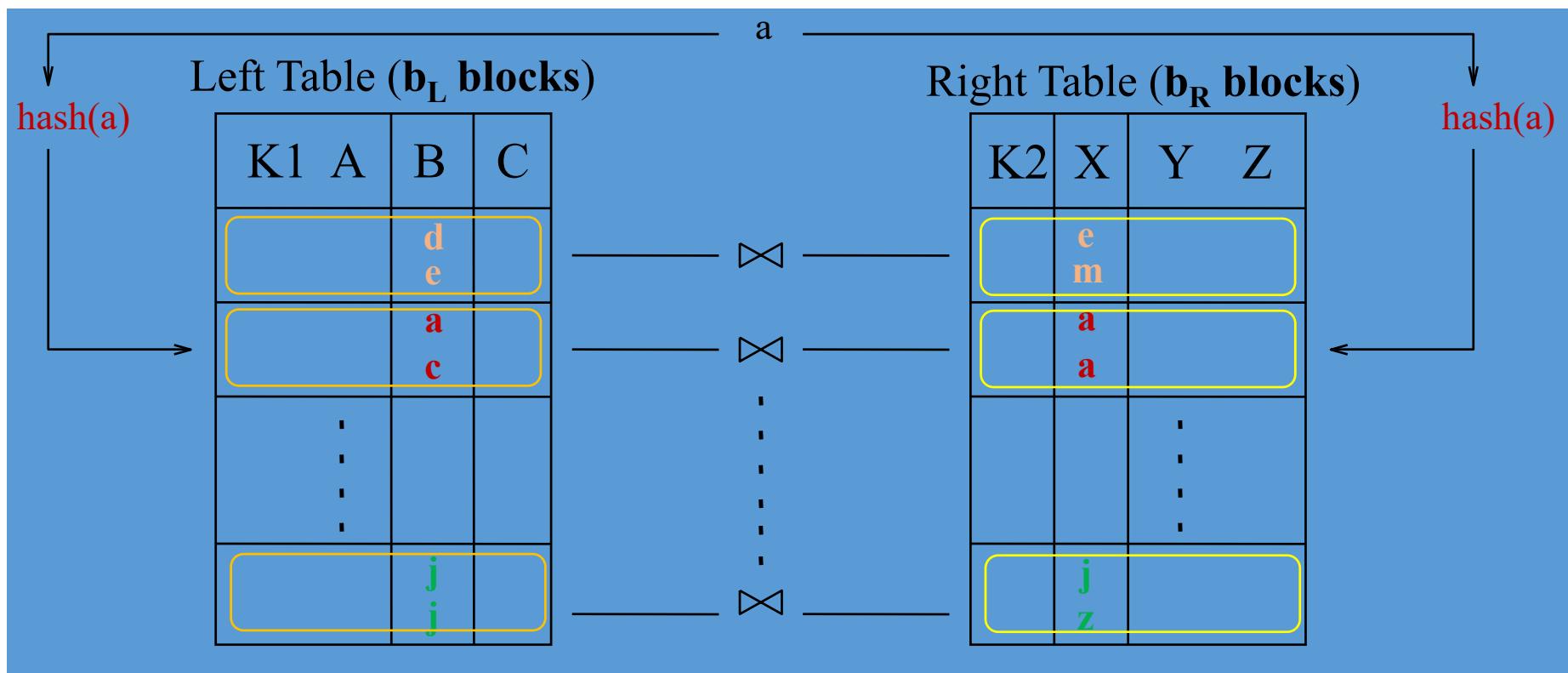


Why 1.8M pointers to follow



Hashed join

- This join is possible only if **both tables are hashed on the same key (join) attribute with the same hash function**
- Find the first/next key value X in the left table, use the hashed access to fetch the tuples with key = X from left and right table, add to the result the joined tuples that match



Cost of hashed joins

- The cost is **linear in the number of blocks of the hash-based structure**
 - If the two hashes are both primary storages:
$$C = b_L + b_R$$
 - Note that the two hashes have the same number of buckets, but the number of blocks b_L and b_R may (slightly) differ due to overflows

COST-BASED OPTIMIZATION

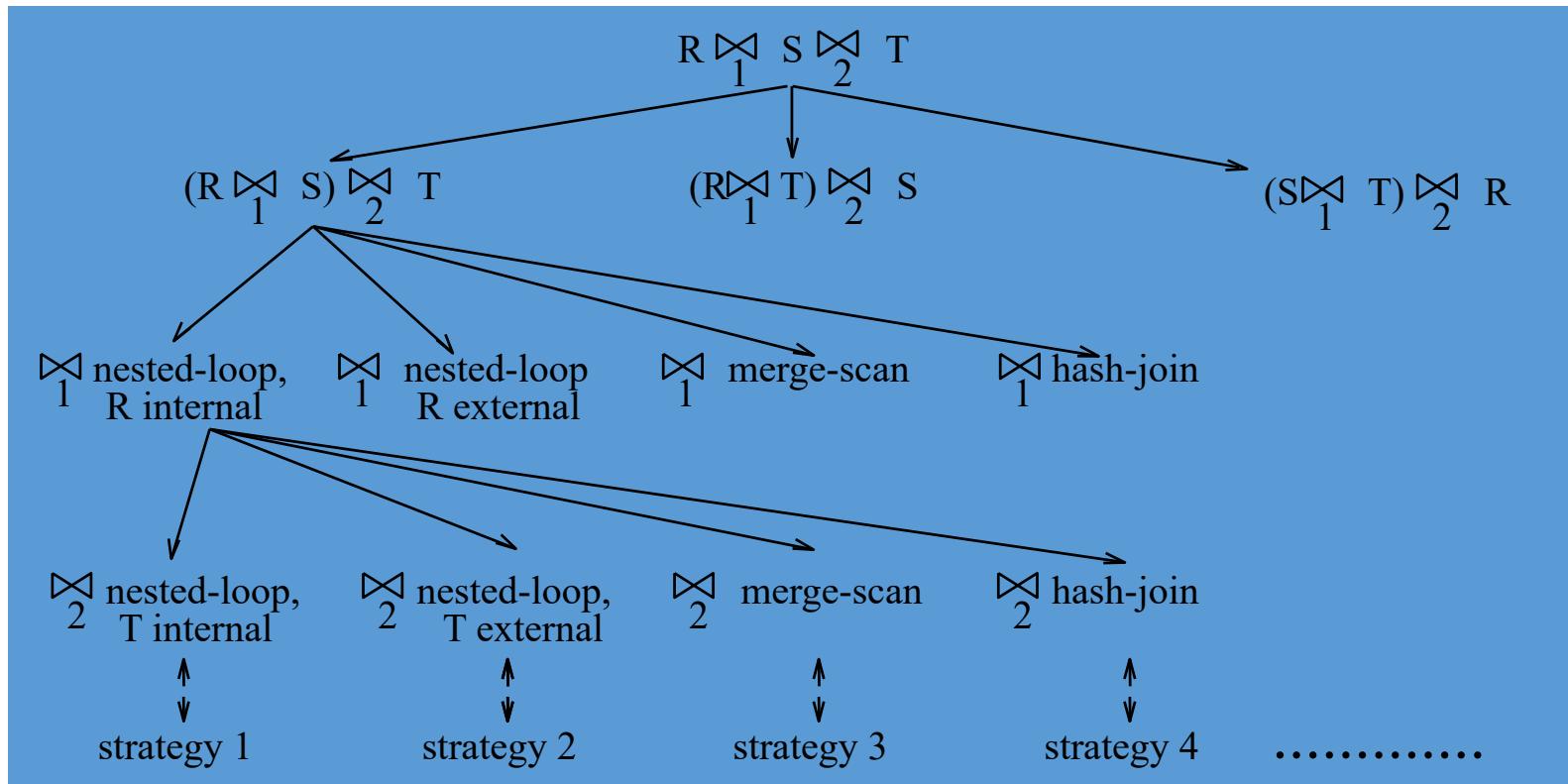
Cost-based optimization

- An optimization problem, whose decisions are:
 - The data access operations to execute
(e.g., scan vs index access)
 - The order of operations (e.g., the join order)
 - The option to allocate to each operation
(e.g., choosing the join method)
 - Parallelism and pipelining can improve performances

Approach to query optimization

- Optimization approach:
 - Make use of profiles and of approximate cost formulas
 - Construct a decision tree, in which
 - each node corresponds to a choice
 - each leaf node corresponds to a specific execution plan

An example of decision tree



Approach to query optimization

- Assign to each plan a cost:
 - $C_{\text{total}} = C_{\text{I/O}} n_{\text{I/O}} + C_{\text{cpu}} n_{\text{cpu}}$
- Choose the plan with the lowest cost, based on operations research (branch and bound)
- Optimizers should obtain ‘good’ solutions in a very short time

Approaches to query execution

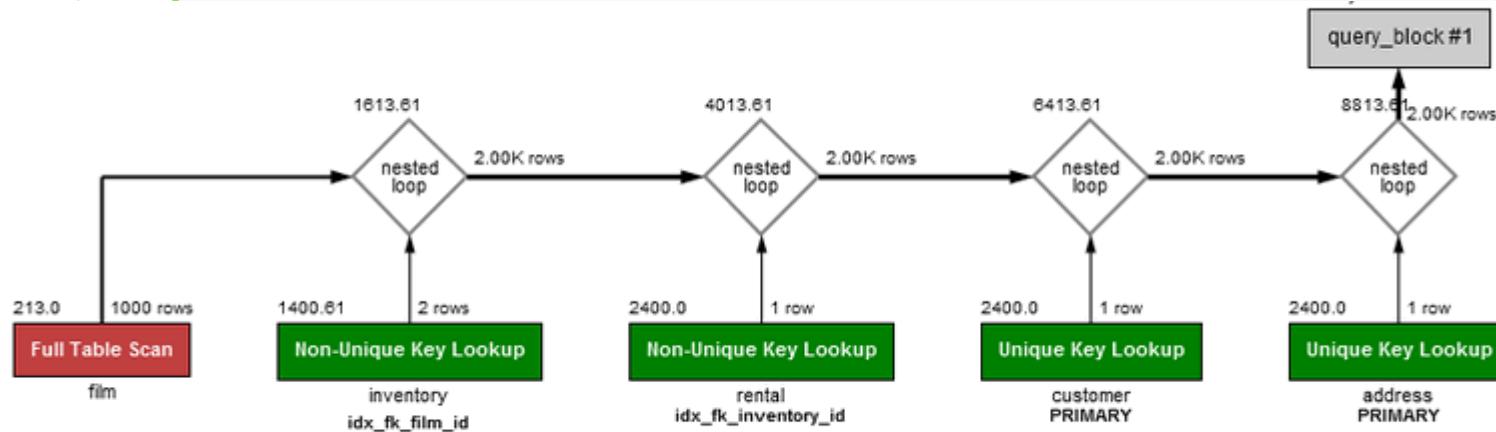
- **Compile and store:** the query is compiled once and executed many times (**prepared statement**)
 - The internal code is stored in the DBMS, together with an indication of the dependencies of the code on the particular versions of catalog used at compile time
 - On relevant changes of the catalog, the compilation of the query is invalidated and repeated
- **Compile and go:** immediate execution, no storage
 - Even if not stored, the code may live for a while in the DBMS and be available for other executions

Summary: Query Optimization

- Important task of DBMSs
- Goal is to minimize # I/O blocks
- Search space of execution plans is huge
- Heuristics based on algebraic transformation lead to good logical plan (e.g., apply first the operations that reduce the size of intermediate results), but no guarantee of optimal plan
- More details in other books (suggested: Elmasry-Navathe)

Query plan in MySQL workbench

```
1  SELECT CONCAT(customer.last_name, ' ', customer.first_name)
2    AS customer, address.phone, film.title FROM rental
3  INNER JOIN customer ON rental.customer_id = customer.customer_id
4  INNER JOIN address ON customer.address_id = address.address_id
5  INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
6  INNER JOIN film ON inventory.film_id = film.film_id
7 WHERE rental.return_date IS NULL
8 AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
9 LIMIT 5;
```



Determine the execution plan of your query

- “EXPLAIN PLAN” SQL statement
- Oracle:
http://docs.oracle.com/cd/B28359_01/server.111/b28274/optimops.htm
- SQLite: <http://www.sqlite.org/eqp.html>
- MySQL:
<https://dev.mysql.com/doc/refman/8.0/en/execon-plan-information.html>
- MS SQL server: [http://msdn.microsoft.com/en-us/library/ms176005\(v=sql.105\).aspx](http://msdn.microsoft.com/en-us/library/ms176005(v=sql.105).aspx)

Application architectures for data integration

Databases 2

Summary

- Goals: understanding database development in the context of architecture and application development
- Two-tier architectures (client-server)
- Three-tier architectures
 - Web: Web server & script engine
- Data integration in three tier architectures
- Requirements for a better integration approach

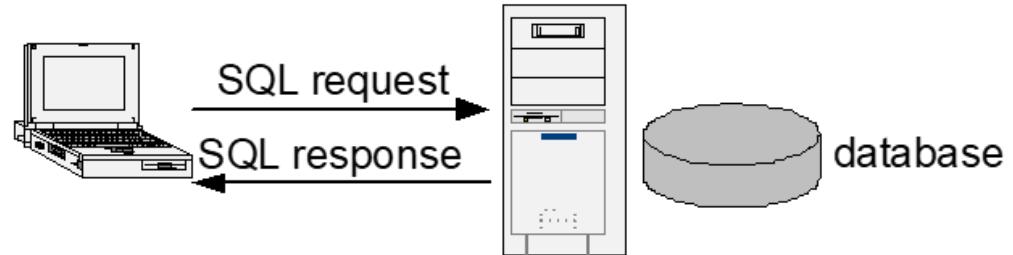
Architecture: definition and classification

- Architecture = mix of HW, SW and network resources
- Classification parameters
 - Type and functionality of HW resources
 - Topology of links connecting HW resources
 - Structure of software modules and relationship between each other

Brief history of distributed architectures

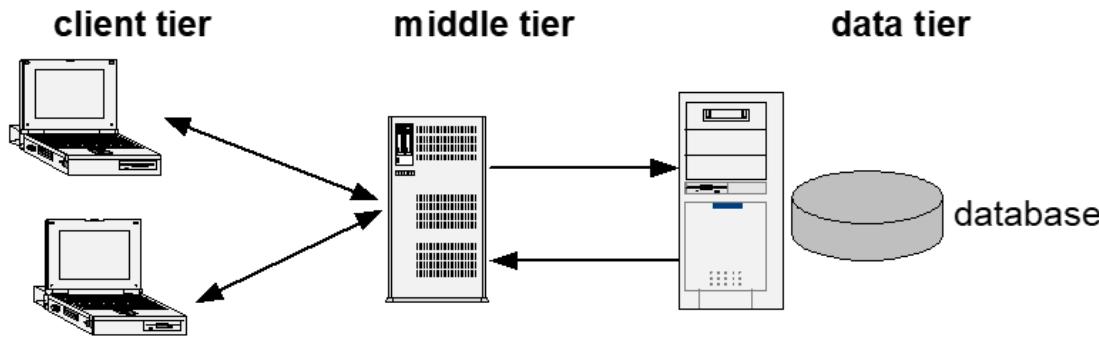
- 60's-70's: One-tier architectures
- 70's-80's: Client-Server architectures
- 80's: Distributed architectures based on Remote Procedure Call (RPC)
- 90's: OO distributed architectures
- Late 90's: Internet and Web based architectures
- Today: Distributed Web based and mobile architectures, service oriented architectures (SOA, REST, Micro Services), Cloud and virtualized architectures, Application Service Provisioning (ASP)

Client server



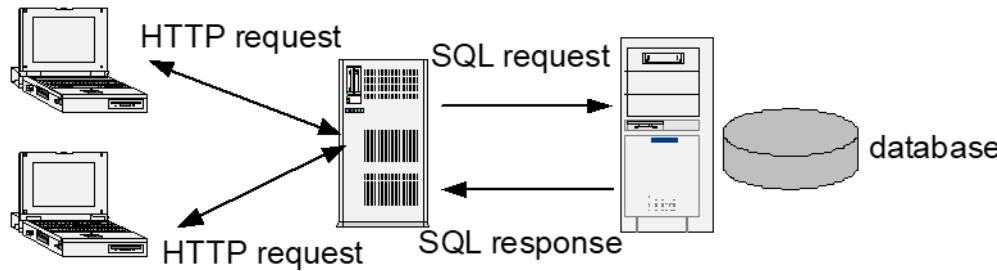
- Specialized HW
 - Server for data management
 - Client for presentation layout
- Topology: LAN with 1 or more servers and N clients
- Software structure: functional partitioning
 - Client SW sends requests to the server (database) by means of SQL queries. Client SW contains both business and presentation logic (e.g. pre-fetch, cache, client-side processing)
 - Server SW processes the query and responds sending the result set back to the client. Server SW only deals with data (e.g. integrity constraints, stored procedures)

Three-tiers architectures



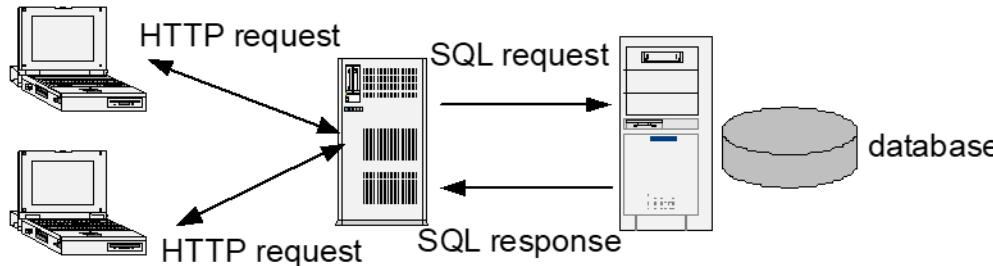
- New element → middle tier
- Adding the middle tier enables to achieve a better separation between the client and the server, since the middle tier:
 - Centralizes connections to the data server
 - Masks data model to the clients
 - Can be replicated to scale up
- This architecture has several variants, depending on the SW features of the middle tier (remote procedure call--RPC, object oriented, message oriented, web)

Web “pure HTML” 3-tiers architectures



- The client is a standard Web browser and it has to cope with the presentation layout only (**thin client**)
- The middle tier
 - includes a Web server that exploits a standard protocol (HTTP)
 - hosts the business logic for dynamically generating content from the raw data of the data tier
 - deals with the presentation layout (it can assemble presentation mark-up, i.e. interfaces)

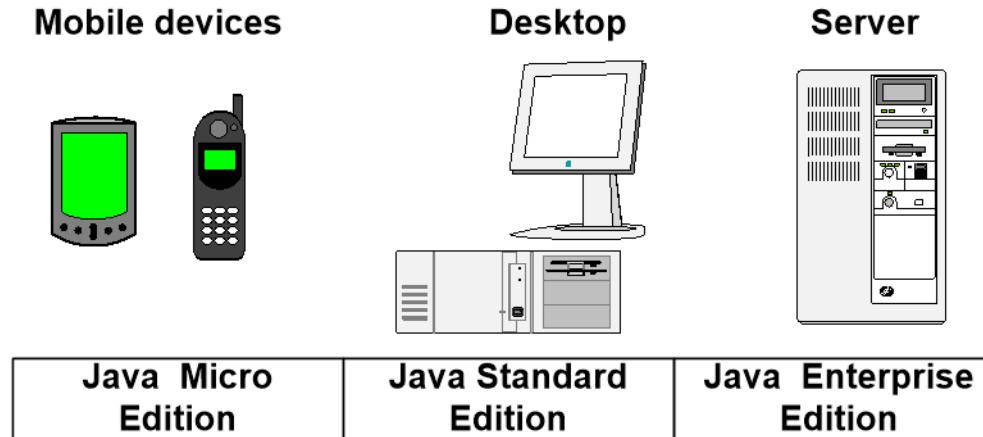
Rich Internet Applications



- The fusion of web and desktop applications
- Enabling technology: client side scripting (with JavaScript)
- **Fat client**: similar to client server but with standard communication protocol (HTTP, Web Socket), language (ECMAScript) and API (DOM, HTML5)
- Supported by the HTML 5 standard
- Features:
 - New interface event types, also specific to touch and mobile apps
 - Asynchronous interaction (AJAX)
 - Client-side persistent data
 - Disconnected / offline applications
 - Native multimedia and 3D support

Three-tier Web applications with Java EE

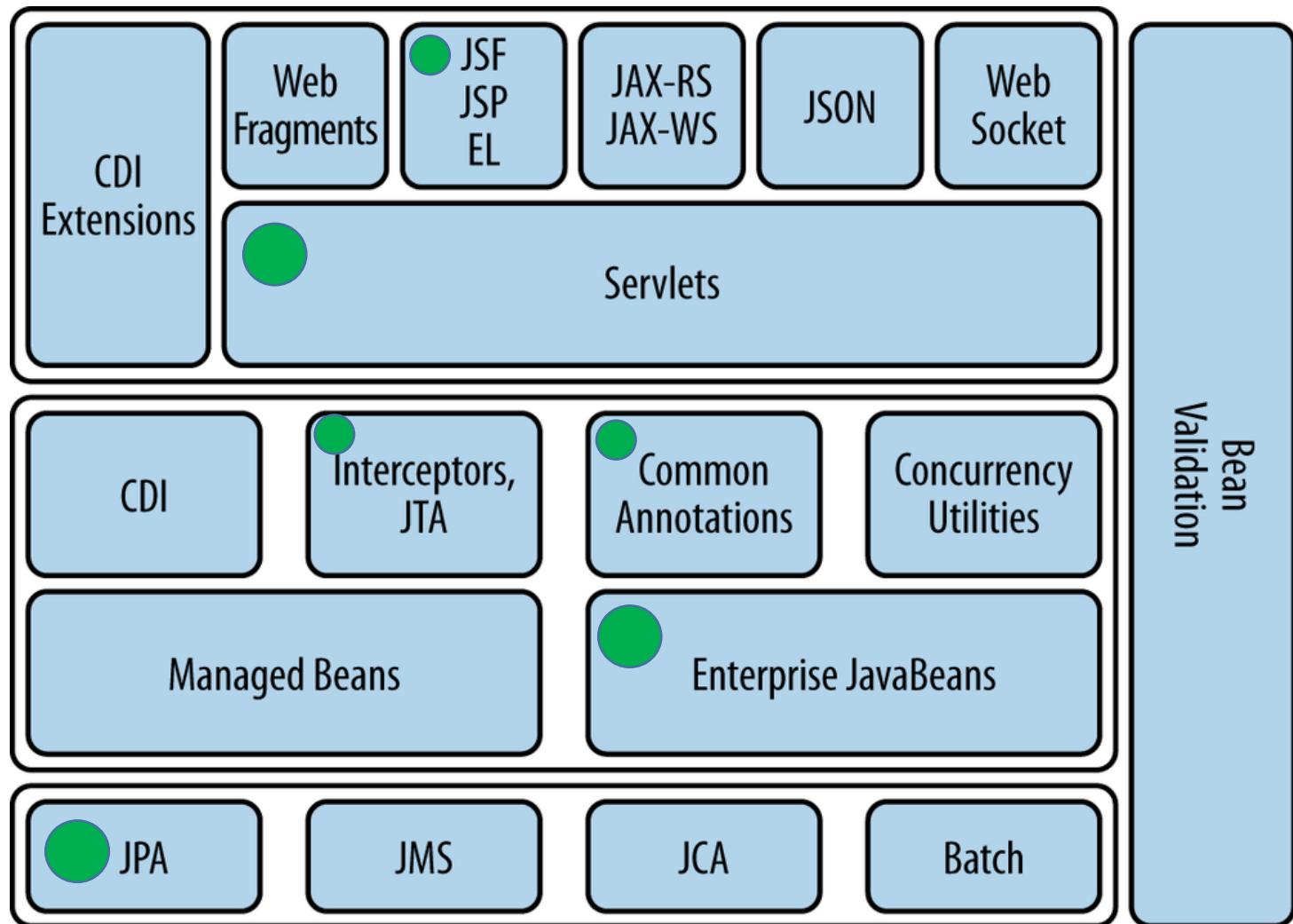
- JEE: platform aimed at the development, release and maintenance of three-tier Web applications
 - API and technology specifications
 - Development and release platforms
 - Reference software implementations
 - Compatibility Test Suite
 - Reference applications (blueprints)



Why JEE?

- Component based development
 - Developing enterprise applications by assembling components, i.e., small self-contained loosely coupled software modules
- Container services
 - Enhancing applications with many functional and non functional requirements provided off the shelf by the environment where components execute (security, transactionality, scalability, failure recovery, interoperability with external systems..)
- Declarative development
 - Declaring what is needed rather than programming it (declarative security, declarative transactions, declarative object to relation mapping)

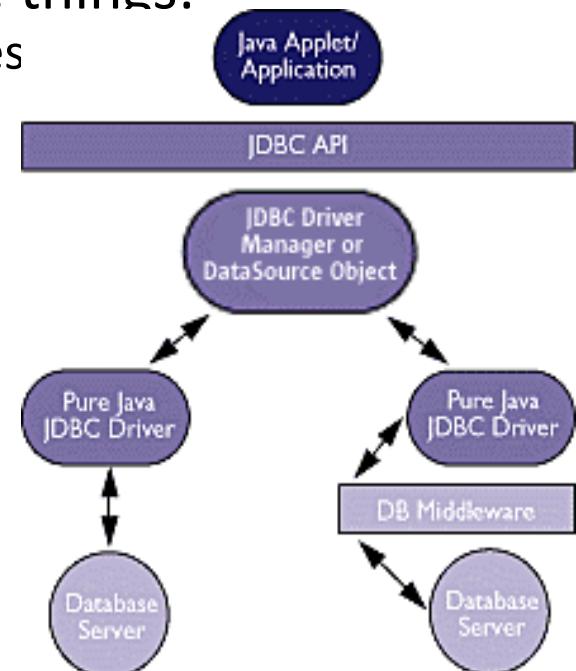
The JEE Stack



In green those used in the optional project

Java DataBase Connectivity (JDBC)

- The JDBC API was the first industry standard for database-independent connectivity between the Java programming language and databases
- The JDBC API makes it possible to do three things:
 - Establish a connection with a database or access source
 - Send SQL statements
 - Process the results
- Still important but superseded by JTA/JPA
- See prerequisite materials on
Beep web site of the course
**Tecnologie Informatiche per il web-
WEB TECHNOLOGIES
[PIERO FRATERNALI]**



Servlet

- Servlets are the Java technology for Web application development (**in the presentation tier**)
 - offer component-based, platform-independent method for building Web-based applications
 - have access to other Java APIs, including the JDBC API to access enterprise databases
 - Executed within a container (**servlet container**), which provides such services as concurrency and lifecycle management
- See prerequisite materials on Beep web site of the course **Tecnologie Informatiche per il web- WEB TECHNOLOGIES [PIERO FRATERNALI]**

Enterprise Java Beans

- Enterprise JavaBeans (EJB) technology is the server-side component architecture of JEE
- Focus on **Business Tier** component development
- Enables development of distributed, transactional, secure and portable applications based on Java technology
- EJB components execute within a container (**EJB container**)
- The EJB container offer services for lifecycle management, transaction management, replication and scaling
- Use advance language features
 - Annotations
 - Dependency injection
- EJB components can be used by the Web front end to interact with the business functions and data access services

Java Persistence API (JPA)

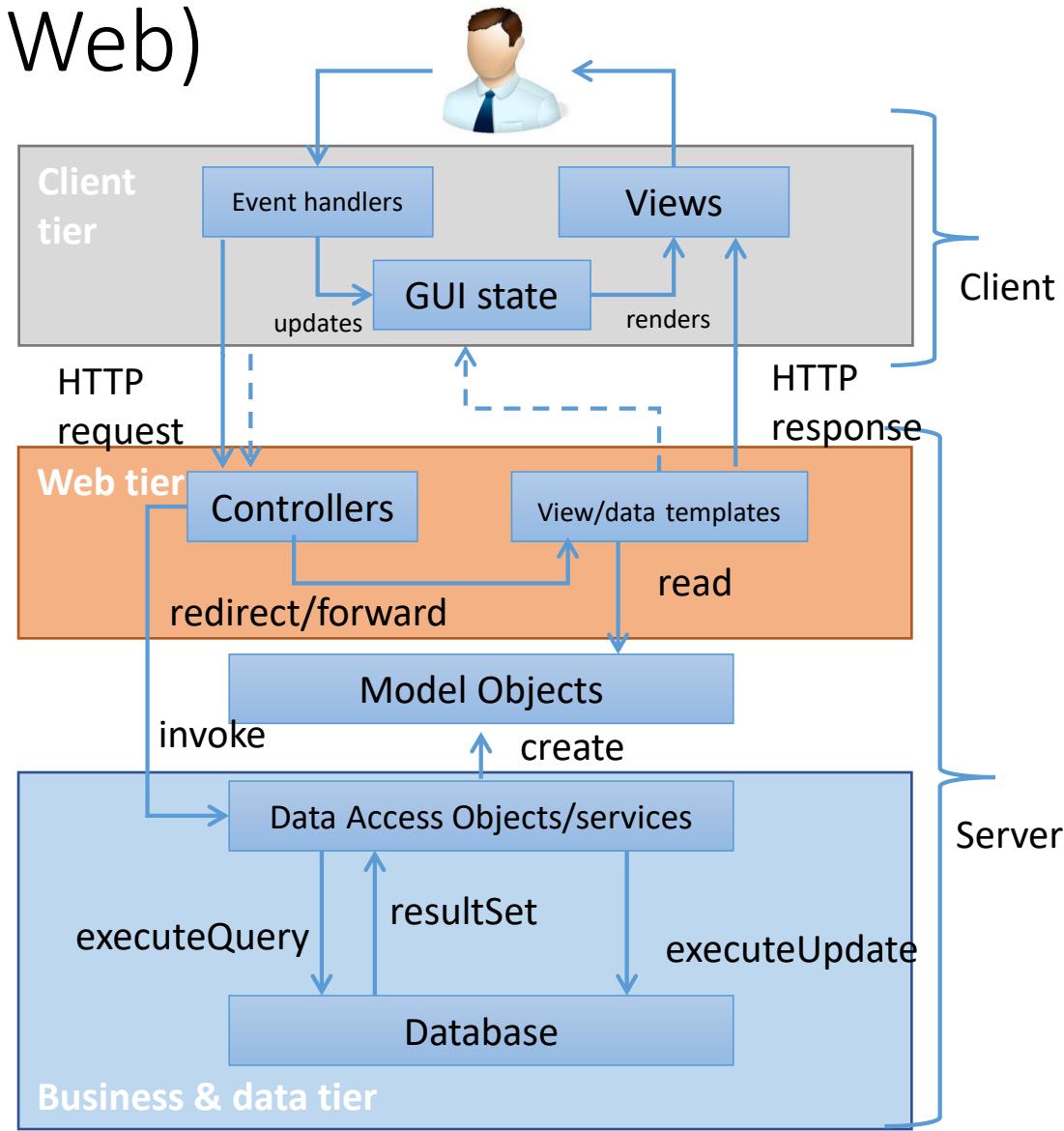
- The specification of an interface for mapping relational data to object oriented data in Java
- Integrated in both Java Standard Edition and Java Enterprise Edition
- Comprises:
 - The API implementation package javax.persistence
 - The Java compatible query language Java Persistence Query Language (JPQL)
 - The specification of the metadata for defining object relational mappings (ORMs)
- Supersedes the use of JDBC

Java Transaction API (JTA)

- An API for managing transactions **in Java**
- It allows a component to start, commit and rollback transactions in a resource-agnostic way
- With JTA Java components can manage multiple resources (i.e. databases, messaging services) in a single transaction with a unique interaction model
- Transactional properties can be expressed
 - declaratively (with `@transactional` code annotation), method by method, eliminating the need to explicit transaction demarcation code
 - Programmatically, with the functions of the `UserTransaction` interface

How to connect an application to the database (3-tier, Web)

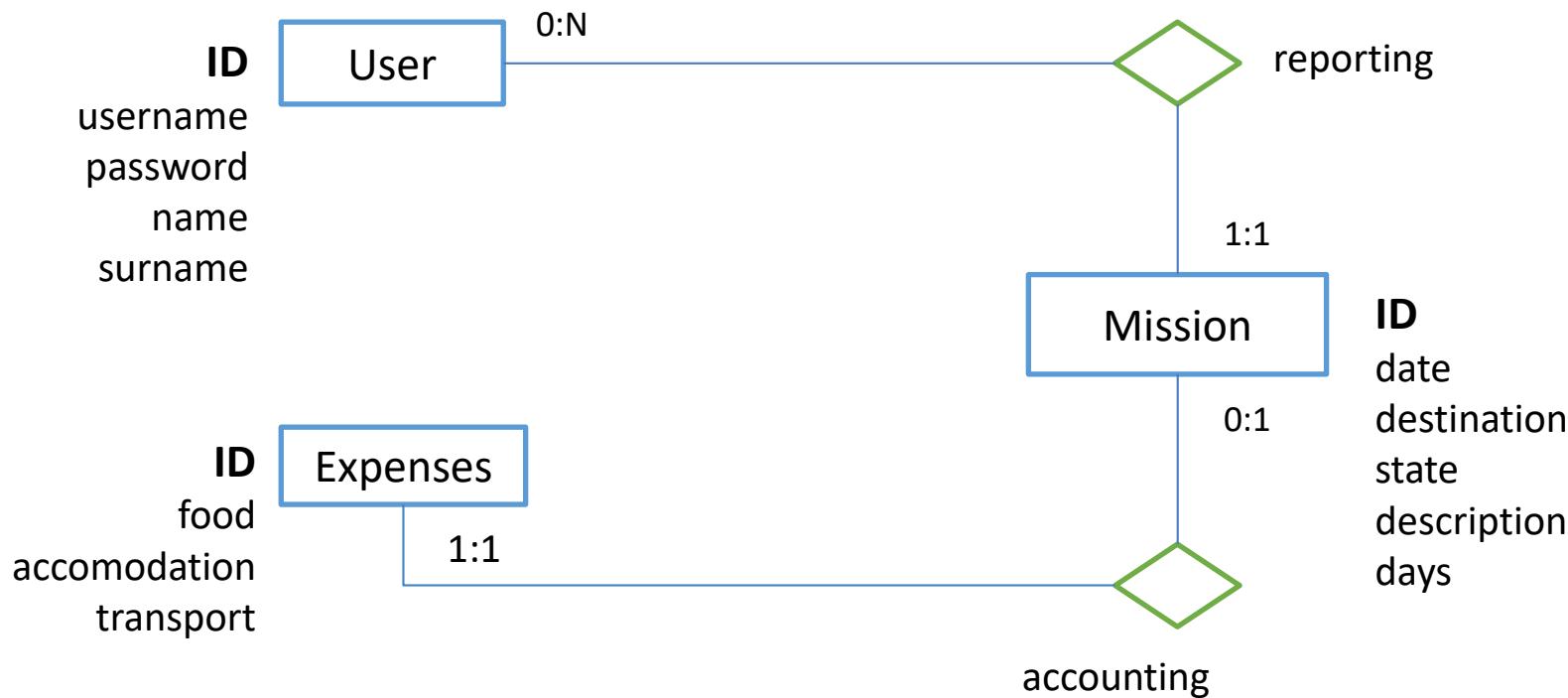
- **Client tier:** handles user's events, invokes the web tier and renders results in the interface (aka view)
- **Web tier:** dispatches (controllers) client calls to business logic components formats and sends back results (extracted from model objects)
- **Business and data tier:** implements business logic and performs data queries and updates within transactions



Case study: expense report

A Web application supports the management of travel expenses. After logging in, the user accesses a HOME page where there is a list of travel missions; a mission belongs to a user and has a date, a place, a description, a number of days of duration, and a status ("open", "reported", "closed"). The list shows the date and place of the missions, which are sorted by date descending. On the HOME page there is a form, with which the user can create a new mission, by entering all the data, which are mandatory. A new mission is always in the "open" state. After creating a mission, the user is returned to the HOME page. When the user selects a mission in the list, a MISSION DETAIL page appears, showing all the mission data. If the mission is in the "open" state, a form appears for entering the expenses incurred during the mission; the form contains three fields: food costs, accommodation costs, transport costs. Sending the form data causes the mission status to change from "open" to "reported", and the return to the MISSION DETAIL page. Total expense should be less than 100€ otherwise the report is rejected with an error. If the mission is in the "reported" status, a CLOSE button appears and the user can click on it to declare that he has received the reimbursement; this causes the mission status to change from "reported" to "closed" and the redisplay of the MISSION DETAIL page. If the mission is in the "closed" status, the MISSION DETAIL page shows the mission data completed with the value of the three types of expenditure.

Database design



Logical database schema

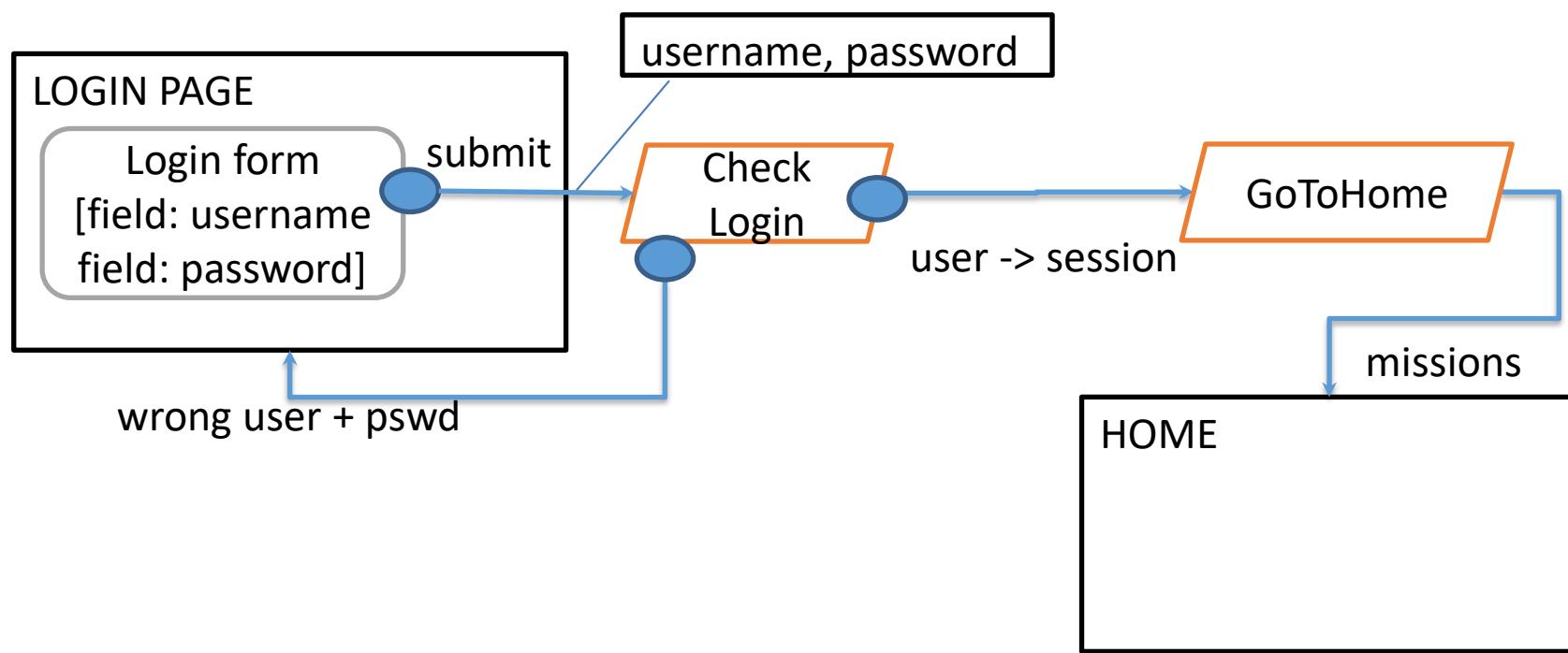
```
CREATE TABLE `user` (
  `id` int(11) NOT NULL
  AUTO_INCREMENT,
  `username` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  `name` varchar(45) NOT NULL,
  `surname` varchar(45) NOT NULL,
  PRIMARY KEY (`id`))
```

```
CREATE TABLE `mission` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `date` date NOT NULL,
  `destination` varchar(45) NOT NULL,
  `state` int(11) NOT NULL DEFAULT '0',
  `description` varchar(45) NOT NULL,
  `days` int(11) NOT NULL,
  `reporter` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `id_reporter` FOREIGN KEY
  (`reporter`) REFERENCES `user` (`id`)
  ON DELETE CASCADE ON UPDATE CASCADE)
```

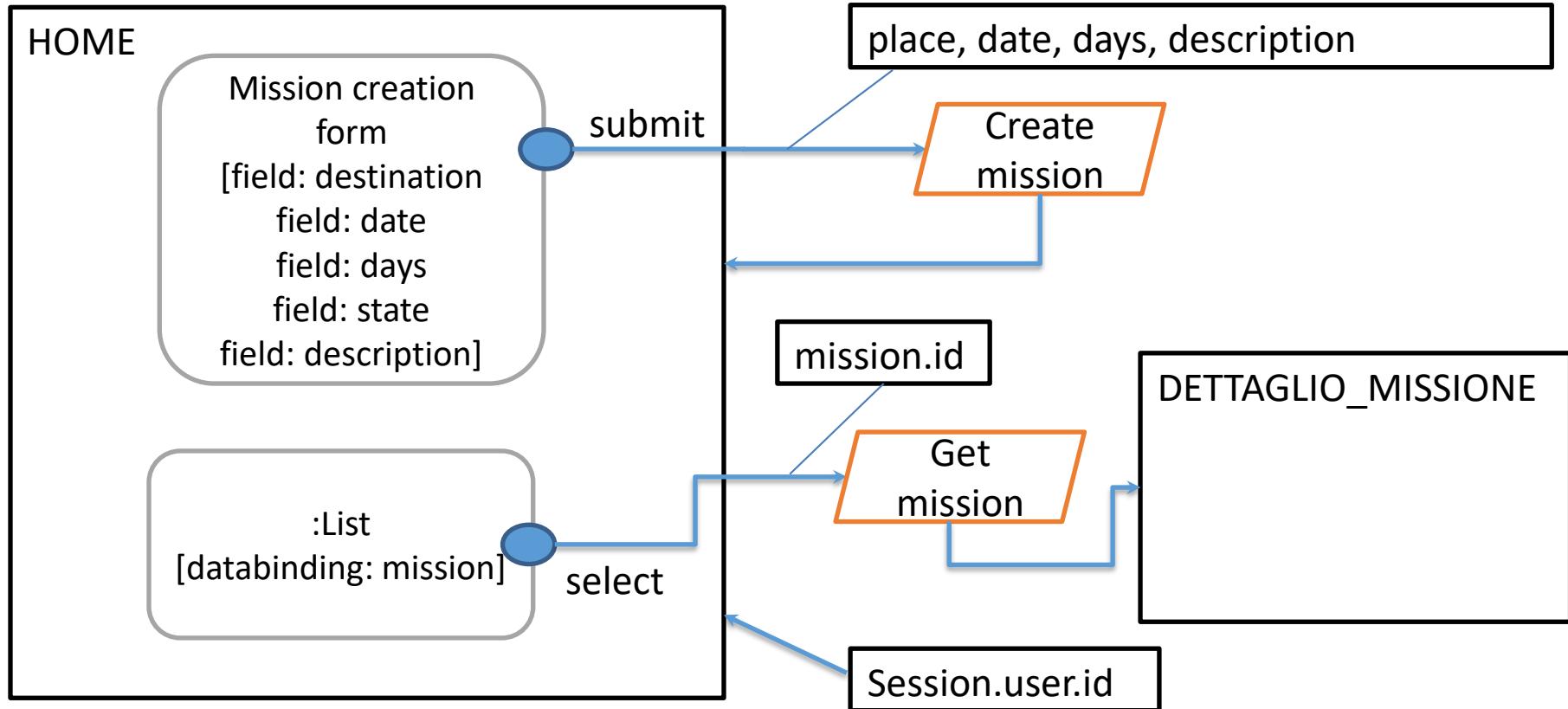
Logical database schema

```
CREATE TABLE `expenses`  
(  
`id` int(11) NOT NULL AUTO_INCREMENT,  
`food` decimal(19,4) NOT NULL,  
`accommodation` decimal(19,4) NOT NULL,  
`transport` decimal(19,4) NOT NULL,  
`mission` int(11) NOT NULL,  
PRIMARY KEY (`id`), KEY `id` (`mission`),  
CONSTRAINT `id` FOREIGN KEY (`mission`) REFERENCES  
`mission`(`id`) ON DELETE CASCADE ON UPDATE CASCADE  
CONSTRAINT `CHK_TotalExp` CHECK ((((`food` +  
`accommodation`) + `transport`) < 100.00))  
)
```

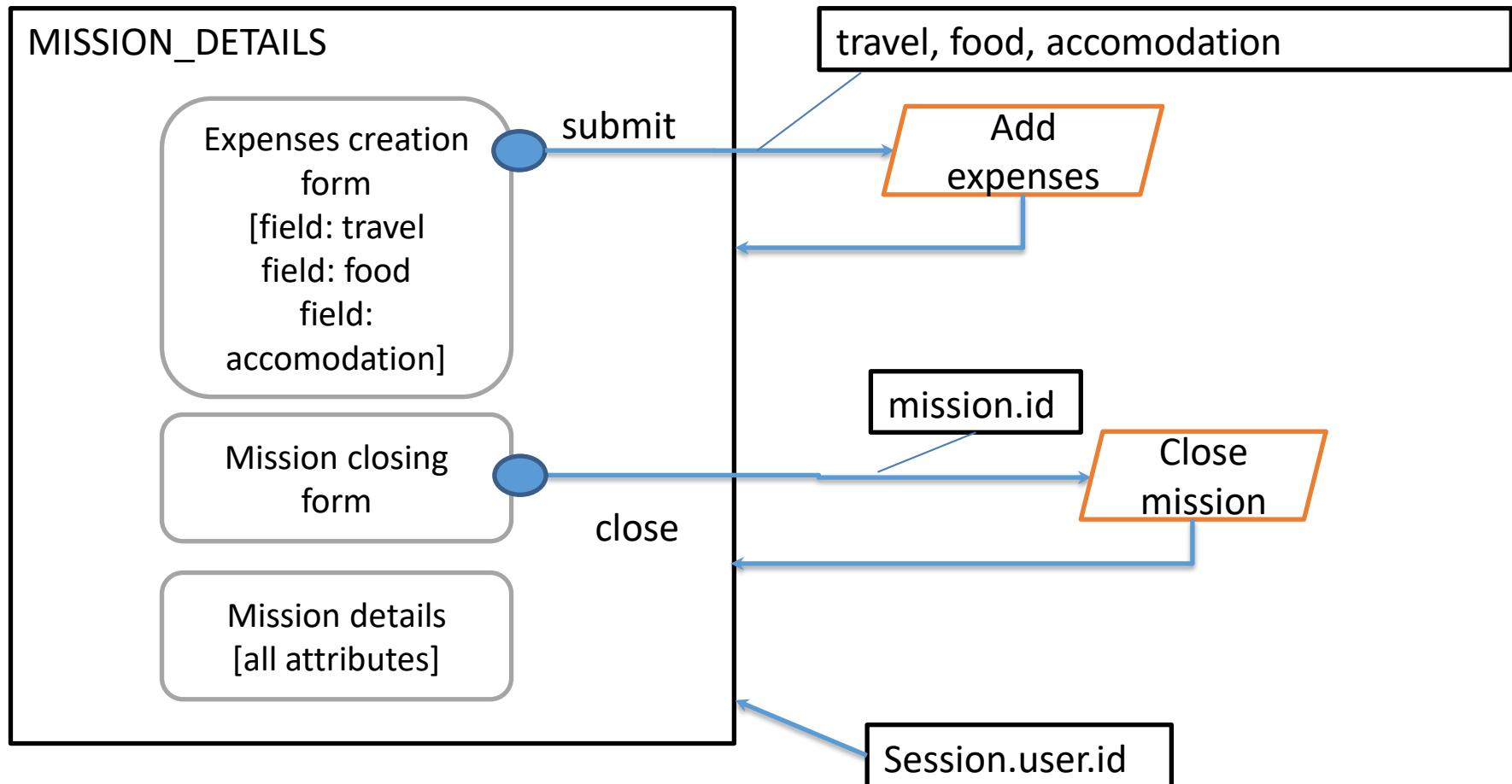
Application design



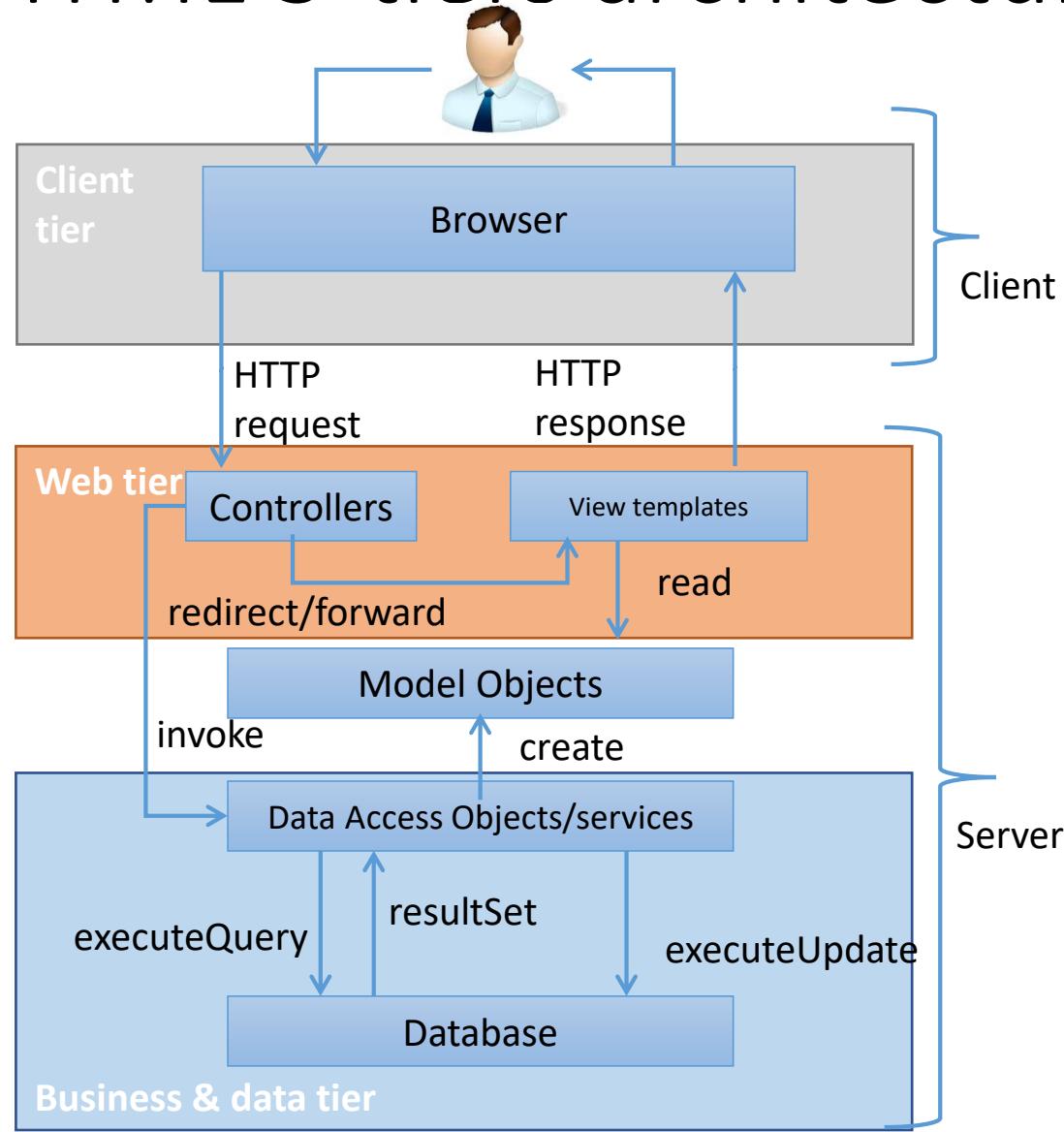
Application design



Application design



Pure HTML 3-tiers architecture



Components

- Model objects (Beans)
 - User
 - Mission
 - ExpenseReport
- Data Access Objects (Classes)
 - UserDao
 - checkCredentials(username, pwd)
 - MissionDAO
 - createMission(mission, userid)
 - findMissionsByUser(userid)
 - findMissionById(missionid)
 - changeMissionStatus(missionid, status)
 - ExpensesDAO
 - addExpenseReport(expenseReport, mission)
 - findExpensesForMission(missionId)
- Controllers (servlets)
 - CheckLogin
 - GoToHomePage
 - CreateMission
 - GetMissionDetails
 - CreateExpenses
 - CloseMission
 - Logout
- Views (Templates)
 - Login
 - Home
 - Mission Details

Database connection in web tier

- Web.xml configuration file in the WEB-INF folder of the web application server

```
<context-param>
    <param-name>dbUrl</param-name>
    <param-value>
        jdbc:mysql://localhost:3306/db_mission
    </param-value>
</context-param>
<context-param>
    <param-name>dbUser</param-name>
    <param-value>piero</param-value>
</context-param>
<context-param>
    <param-name>dbPassword</param-name>
    <param-value>fraternali</param-value>
</context-param>
<context-param>
    <param-name>dbDriver</param-name>
    <param-value>
        com.mysql.cj.jdbc.Driver</param-value>
</context-param>
```

- Utility functions called by all controllers that need be connected to the database

```
public class ConnectionHandler {

    public static Connection getConnection(ServletContext
        context) throws UnavailableException {
        Connection connection = null;
        try {
            String driver = context.getInitParameter("dbDriver");
            String url = context.getInitParameter("dbUrl");
            String user = context.getInitParameter("dbUser");
            String password =
                context.getInitParameter("dbPassword");
            Class.forName(driver);
            connection = DriverManager.getConnection(url, user,
                password);
        } catch (ClassNotFoundException e) {
            throw new UnavailableException("Can't load driver");
        } catch (SQLException e) {
            throw new UnavailableException("Couldn't get db
                connection");
        }
        return connection;
    }

    public static void closeConnection(Connection
        connection) throws SQLException {
        if (connection != null) { connection.close();}
    }
}
```

(Repetitive) code in the controllers

- At initialization

```
public void init() throws ServletException {  
    connection = ConnectionHandler.getConnection(  
        getServletContext());  
}
```

- At termination

```
public void destroy() {  
    try {  
        ConnectionHandler.closeConnection(connection);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

- Every controller repeats the same code for acquiring and releasing the connection
- Connections are not handled efficiently (one per controller, stored as a data member of the controller class)
- Forgetting to release a connection is costly

Persistent data: extraction

```
// Method of the business object MissionDAO
public List<Mission> findMissionsByUser(int userId) throws SQLException {
    List<Mission> missions = new ArrayList<Mission>();
    String query = "SELECT * from mission where reporter = ? ORDER BY date DESC";

    try (PreparedStatement pstatement = connection.prepareStatement(query)) {
        pstatement.setInt(1, userId);
        try (ResultSet result = pstatement.executeQuery()) {
            while (result.next()) {
                Mission mission = new Mission();
                // Copy cursor data into Java bean objects...
                mission.setId(result.getInt("id"));
                mission.setStartDate(result.getDate("date"));
                mission.setDestination(result.getString("destination"));
                mission.setStatus(result.getInt("status"));
                mission.setDescription(result.getString("description"));
                mission.setDays(result.getInt("days"));
                mission.setReporterID(userId);
                missions.add(mission);
            }
        }
    } // exception handling omitted for brevity
    return missions;
}
```

Note: JDBC type is ResultSet
Application type is List<Mission>

Persistent data: creation

```
// Method of the business object MissionDAO

public void createMission(Date startDate, int days, String destination,
                           String description, int reporterId)
throws SQLException {

    String query = "INSERT into mission (date, destination, status,
                                         description, days, reporter) VALUES (?, ?, ?, ?, ?, ?)";
    try (PreparedStatement pstatement =
         connection.prepareStatement(query)) {
        // Copy Java parameters into SQL binding variables
        pstatement.setDate(1, new java.sql.Date(startDate.getTime()));
        pstatement.setString(2, destination);
        pstatement.setInt(3, MissionStatus.OPEN.getValue());
        pstatement.setString(4, description);
        pstatement.setInt(5, days);
        pstatement.setInt(6, reporterId);
        pstatement.executeUpdate();
    }
}
```

Persistent data: modification

```
// Method of the missionDAO business object
public void changeMissionStatus(int missionId,
    MissionStatus missionStatus) throws SQLException {

String query = "UPDATE mission SET status = ? WHERE id = ? ";
try (PreparedStatement pstatement =
    connection.prepareStatement(query))
{
    // copy the data to modify into the SQL query
    pstatement.setInt(1, missionStatus.getValue());
    pstatement.setInt(2, missionId);
    pstatement.executeUpdate();
}
}
```

- Note the use of the `try with resource` clause to avoid the manual management of JDBC objects, such as statements and result sets

Transactions

```
// Method of expenseDAO business object
public void addExpenseReport(ExpenseReport expenseReport, Mission mission)
    throws SQLException, BadMissionForExpReport {
    // Check that the mission exists and is in OPEN state
    ...
    MissionsDAO missionDAO = new MissionsDAO(connection);
    String query = "INSERT into expenses (food, accomodation, transport, mission)
        VALUES (?, ?, ?, ?)";  

        //Delimit the transaction explicitly
connection.setAutoCommit(false); //Override default commit after each statement
try (PreparedStatement pstatement = connection.prepareStatement(query)) {
    pstatement.setDouble(1, expenseReport.getFood());
    pstatement.setDouble(2, expenseReport.getAccomodation());
    pstatement.setDouble(3, expenseReport.getTransportation());
    pstatement.setInt(4, expenseReport.getMissioId());
    pstatement.executeUpdate(); // 1st update
        // 2nd update, to be executed atomically
    missionDAO.changeMissionStatus(expenseReport.getMissioId(),
        MissionStatus.REPORTED);
    connection.commit();
} catch (SQLException e) {
    connection.rollback(); // if update 1 OR 2 fails, roll back all work
    throw e;
}
} finally {connection.setAutoCommit(true); // reset to standard
}
```

Wouldn't it be better if

- A database query returned results already encoded in the right application type
- Creating an object would create a tuple without the need of copying data manually
- Updating an object would modify the matching tuple automatically
- Connections and data access objects would be dispatched to the components that need them transparently to the application
- Methods calls could join transactions automatically based on the need

Problem-solution matrix

Requirement	Technology	How to
Avoiding the copy of data from query result cursor to native application types	JPA Object Relational Mapping	Java classes and relational tables can be associated via a declarative mapping
Avoiding the copy of data from program to UPDATE and INSERT SQL statement	JPA Object Relational Mapping	Creation and changes to Java objects can be persisted automatically
Avoiding the manual management of connections	JTA transactions EJB dependency injection	Connection object masked by higher level objects (EntityManager) automatically injected into the components that need them
Avoiding manual demarcation of transactions	JTA transactions EJB container managed transactions	Global transactions managed by the container and propagated to object methods that can “join” them

Data extraction with JPA

```
// method of missionService EJB

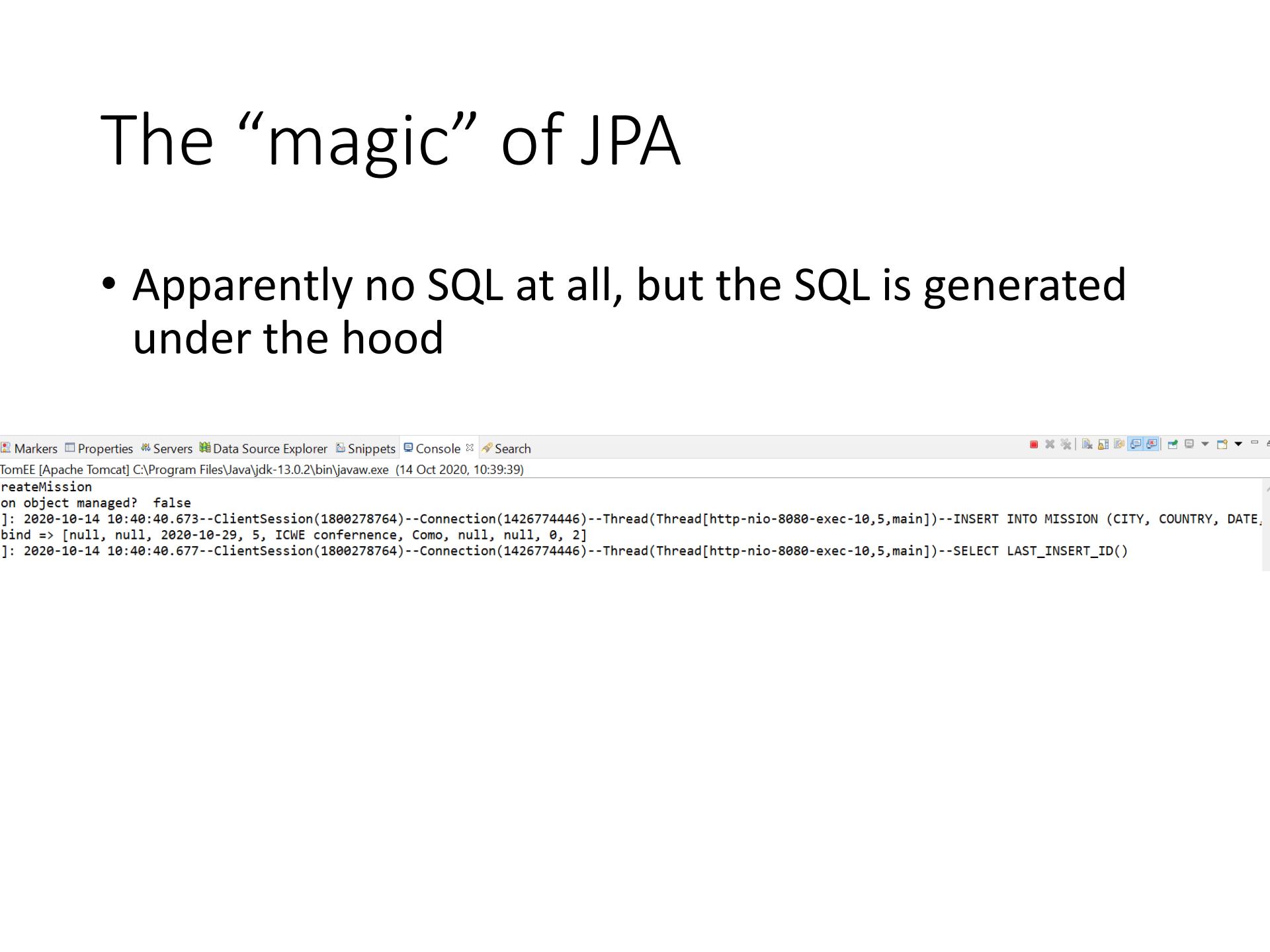
public List<Mission> findMissionsByUser(int userId) {
    Reporter reporter = em.find(Reporter.class, userId);
    List<Mission> missions = reporter.getMissions();
    return missions;
}
```

Data creation with JPA

```
public void createMission(Date startDate, int days, String destination, String description, int reporterId) {  
    Reporter reporter = em.find(Reporter.class, reporterId);  
Mission mission = new Mission(startDate, days, destination,  
                                description, reporter);  
reporter.addMission(mission);  
em.persist(reporter);  
}
```

The “magic” of JPA

- Apparently no SQL at all, but the SQL is generated under the hood



A screenshot of an IDE interface, likely Eclipse or IntelliJ IDEA, showing a Java application running on TomEE. The top menu bar includes 'Markers', 'Properties', 'Servers', 'Data Source Explorer', 'Snippets', 'Console', and 'Search'. The console tab shows the following log output:

```
TomEE [Apache Tomcat] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (14 Oct 2020, 10:39:39)
createMission
on object managed? false
]: 2020-10-14 10:40:40.673--ClientSession(1800278764)--Connection(1426774446)--Thread(Thread[http-nio-8080-exec-10,5,main])--INSERT INTO MISSION (CITY, COUNTRY, DATE,
bind => [null, null, 2020-10-29, 5, ICWE conference, Como, null, null, 0, 2]
]: 2020-10-14 10:40:40.677--ClientSession(1800278764)--Connection(1426774446)--Thread(Thread[http-nio-8080-exec-10,5,main])--SELECT LAST_INSERT_ID()
```

Data modification with JPA

```
// Method of the missionService business object
public void reportMission(int missionId,
                           MissionStatus missionStatus) {
    Mission mission = em.find(Mission.class, missionId);
    mission.setStatus(MissionStatus.REPORTED);
}
```

Transaction management in JPA

```
public void addExpenseReport(Expense expenseReport, int missionId)
throws BadMissionForExpReport {

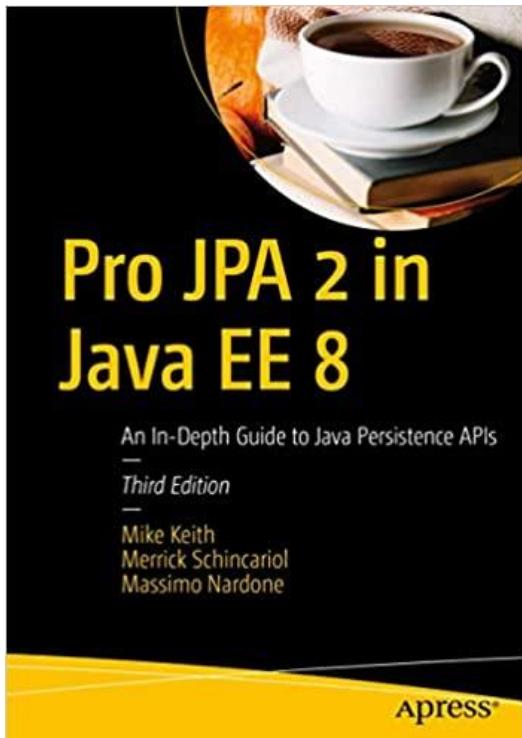
// Check that the mission exists and is in OPEN state
Mission mission = em.find(Mission.class, missionId); // now mission is managed

if (mission == null | mission.getStatus() != MissionStatus.OPEN) {
throw new BadMissionForExpReport("Mission cannot introduce expense report");
}

try { // this code is transactional!
mission.setStatus(MissionStatus.REPORTED);
mission.setExpense(expenseReport);
em.persist(mission);
} catch (PersistenceException e) {
e.printStackTrace(); // used for debugging
}
}
```

References

- The Java EE documentation, available at:
 - <https://docs.oracle.com/javaee/7/index.html>
- Bibliography textbook, Chapter 3 and Chapter 6



Java Persistence API Object Relational Mapping

Databases 2

Application data integration

- The end-point of every web application is the DBMS
 - In many cases the database exists before and independently of the applications that use it
 - Moving data back and forth between a DBMS and the object model is **harder** than it needs to be
 - **A lot of repetitive code** is spent to convert row and column data into objects

Object Model vs. Relational Model

- The technique of bridging the gap between the object model and the relational model is known as **object-relational mapping (ORM)**
- ORM techniques try to map the concepts from one model onto another
 - **Impedance mismatch:** The challenge of mapping one model to the other lies in the concepts in one model for which there is no logical equivalent in the other
- A **mediator** is needed to manage the automatic transformation of one into the other

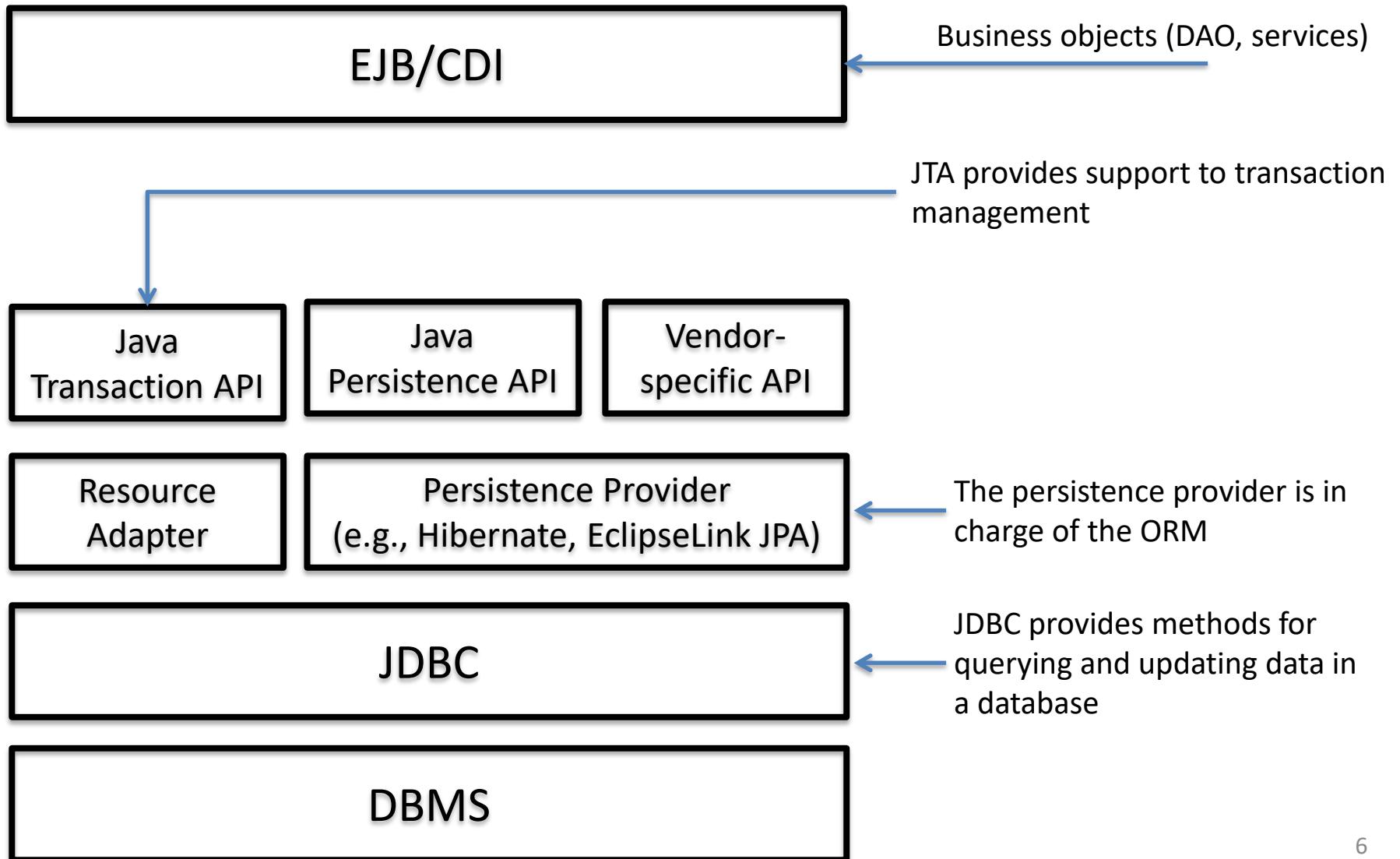
Differences between OM and RM

Object Oriented Model (Java)	Relational Model
Objects, classes	Tables, rows
Attributes, properties	Columns
Identity (physical memory address)	Primary key
Reference to other entity	Foreign key
Inheritance/Polymorphism	Not supported
Methods	Stored procedures, triggers
Code is portable	Not necessarily portable (depending on the vendor)

Java Persistence API

- The **Java Persistence API** bridges the gap between object-oriented domain models and relational database systems
 - JPA provides a POJO (Plain Old Java Object) persistence model for object-relational mapping
- Adds up to other previous proposals
 - JDO: Java data Objects (JDO 2.0: JSR 243)
 - JDBC: Java database connectivity (JDBC 3.0 API)
- Developed as part of [JSR-317](#)
 - In addition to support within EJB, JPA can be used in a standalone Java SE environment
 - Usable with / without a container

JPA Architecture



JPA in a nutshell

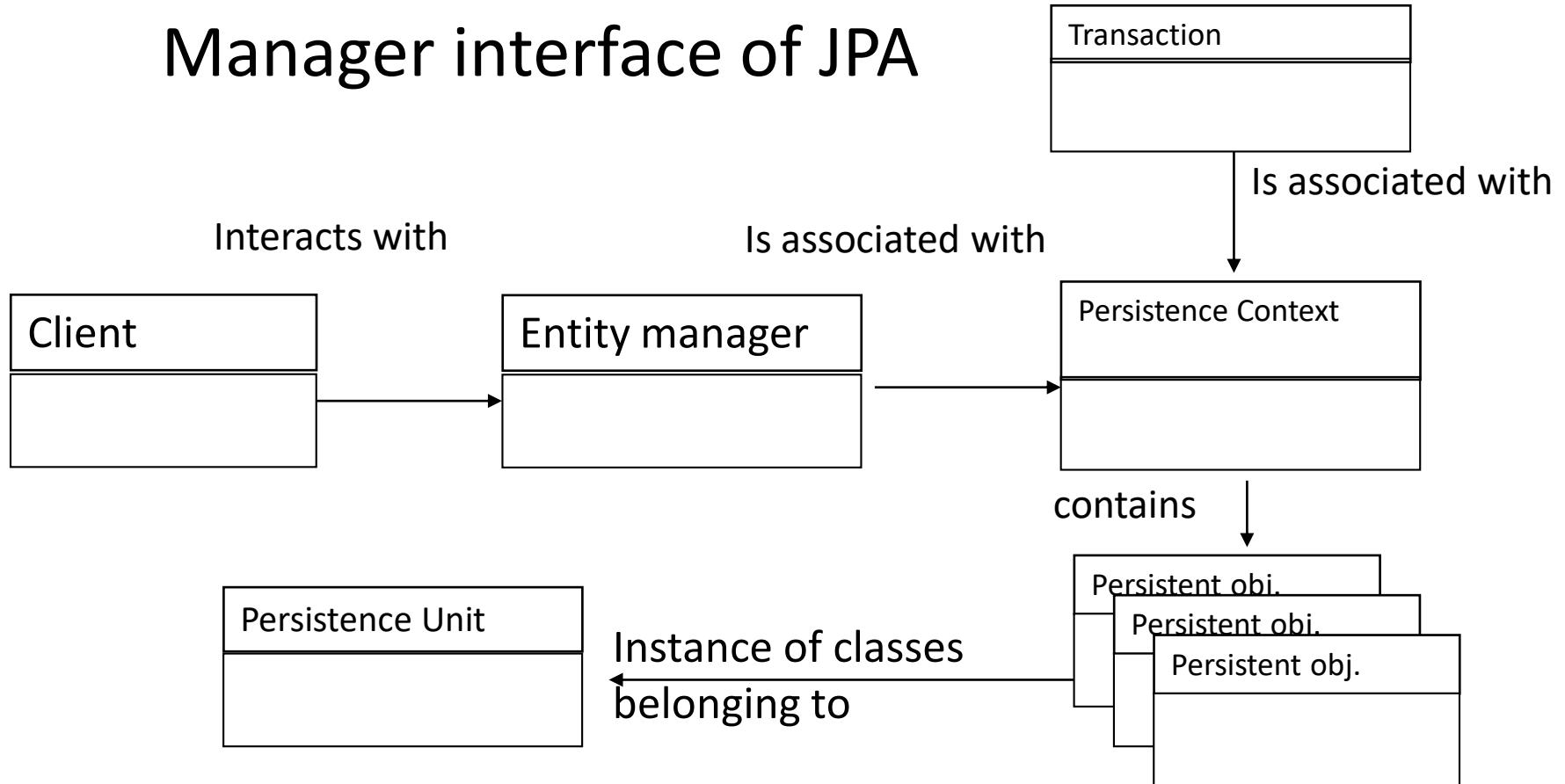
- Java Persistence API main features:
 - **POJO Persistence:** there is nothing special about the objects being persisted, any existing non-final object with a default constructor can be persisted
 - **Non-intrusiveness:** the persistence API exists as a separate layer from the persistent objects, i.e., the mapped objects are not aware of the persistence layer
 - **Object queries:** a powerful query framework offers the ability to query across entities and their relationships without having to use concrete foreign keys or database columns

JPA main concepts

- **Entity**: a class (JavaBean) representing a collection of persistent objects mapped onto a relational table
- **Persistence Unit**: the set of all **classes** that are persistently mapped to **one** database (analogous to the notion of **db schema**)
- **Persistence Context**: the set of all **managed objects** of the entities defined in the persistence unit (analogous to the notion of **db instance**)
- **Managed entity**: an entity part of a persistence context for which the changes of the state are tracked
- **Entity manager**: the interface for interacting with a Persistence Context
- **Client**: a component that can interact with a Persistence Context, indirectly through an Entity Manager (e.g., an EJB component)

How to work with entities

- Entities are accessed through the Entity Manager interface of JPA



Entity Manager interface

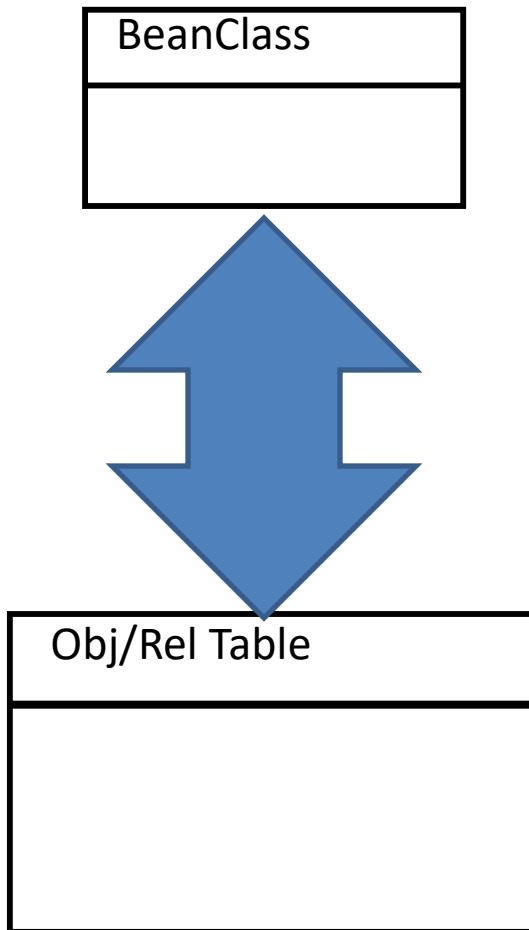
- The Entity Manager exposes all the operations needed to synchronize the managed entities in the persistence context to the database

Method signature	Description
<code>public void persist(Object entity);</code>	Persists an entity instance in the database
<code>public <T> T find(Class<T> entityClass, Object primaryKey);</code>	Finds an entity instance by its primary key
<code>public void remove(Object entity);</code>	Removes an entity instance from the database
<code>public void refresh(Object entity);</code>	Resets the entity instance from the database
<code>public void flush();</code>	Writes the state of entities to the database immediately

Entity

- A Java Bean (POJO Plain Old Java Object) that gets associated to a tuple in a database
 - The class maps to the **table**
 - The objects map to the **tuples**
- The persistent counterpart of an entity has a life longer than that of the application
- The entity class must be associated with the database table it represents (**mapping**)
- An entity **can** enter in a **managed** state, where all the modifications to the object's state are tracked and automatically synchronized to the database

Entity Properties



- Identification (primary key)
- Nesting
- Relationship
- Referential integrity (foreign key)
- Inheritance

Entity (example)

Employee.java

```
@Entity ←  
public class Employee {  
    @Id private int id;  
    private String name;  
    private long salary;  
    public Employee() {}  
    public Employee(int id) { this.id = id; }  
    public int getId() { return id; }  
    public void setId(int id) { this.id = id; }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public long getSalary() { return salary; }  
    public void setSalary (long salary) { this.salary = salary; }  
}
```

The entity instance is just a POJO

Java annotations are used to qualify the class as an entity

Entity constraints

- Entities must respect the following requirements:
 - The entity class must have a public or protected no-arg constructor
 - The entity class must not be final
 - No method or persistent instance variables of the entity class may be final
 - If an entity instance is to be passed by value as a detached object, the Serializable interface must be implemented
- The persistent state of an entity is represented by instance variables, which correspond to JavaBean properties (i.e., with setter and getter methods)

Entity Identification

- In database, objects and tuples have an identity (**primary key**)
- → an entity assumes the identity of the persistent data it is associated to
- **Simple** Primary key = persistent field of the bean used to represent its identity
- **Composite** Primary key = set of persistent fields used to represent its identity
- **Remark:** with respect to the POJOs, the persistent identity is a new concept. **POJOS do not have a durable identity**

Entity identification syntax

```
@Entity  
public class Mission  
    implements Serializable {  
  
    @Id  
    private int id;  
    private String city;  
    ...}
```

- `@Id` tags a field as the simple primary key
- Composite primary keys are denoted using the `@EmbeddedId` and `@IdClass` annotations

Entity: Identifier generation

- Sometimes, applications do not want to explicitly manage uniqueness of data values
 - The persistence provider can automatically generate an identifier for every entity instance of a given type
- This persistence provider's feature is called **identifier generation** and is specified by the @GeneratedValue annotation

Identifier generation options

- Applications can choose one of four different ID generation strategy

Id generation strategy	Description
AUTO	The provider generates identifiers by using whatever strategy it wants
TABLE	Identifiers are generated according to a generator table
SEQUENCE	If the underlying DB supports sequences, the provider will use this feature for generating IDs
IDENTITY	If the underlying DB supports primary key identity columns, the provider will use this feature for generating IDs

Identifier generation annotation

```
@Entity  
public class Mission implements Serializable {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int id;  
  
    private String city;
```

- Whenever possible let the database generate the unique Ids

Attribute specifications

- Attributes can be qualified with properties that direct the mapping between POJOs and relational tables
- Mapping base and special types
 - Large objects
 - Enumerated types
 - As Java enumerations
 - As strings
 - Temporal types
 - Java temporal types
 - JDBC temporal types
- Specifying fetch policy
 - LAZY
 - EAGER (default)

```
@Entity
public class Mission implements
Serializable {

    @Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
private int id;

    @Temporal(TemporalType.
DATE)
private Date date;

    private MissionStatus status;

    @Basic(fetch=FetchType.LAZY)
@LOB
private byte[] photo;
}
```

- Specifying LAZY policy makes the attribute value remain empty when the object is retrieved, until the attributed is accessed explicitly (→ use for LOBs only)

Object to table mapping

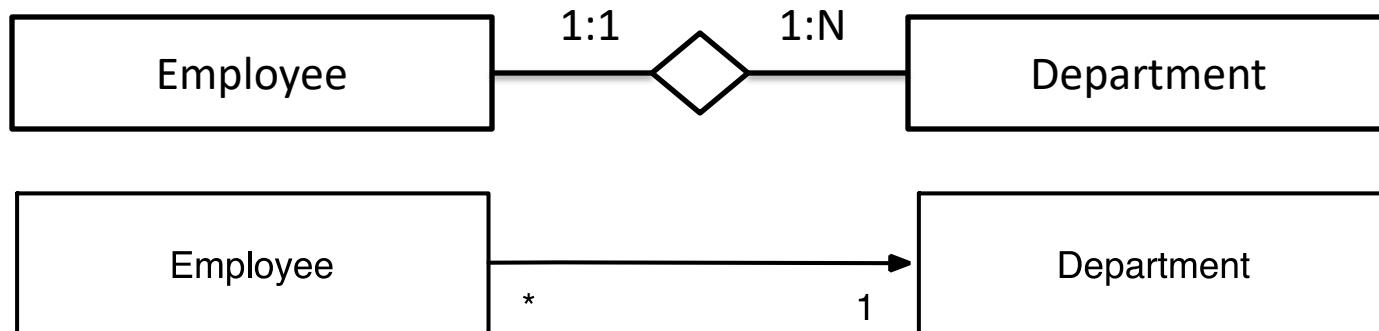
- By default entities are mapped to tables with the same name and their fields to columns with the same names
- Defaults can be overridden
- Some annotations (e.g., @column) may have attributes used for generating the database schema from the entity classes (e.g., nullable)
- Entities may also have attributes that are not persisted (@Transient)

```
@Entity @Table(name="T_BOOKS")
public class Book {

    @Column(name="BOOK_TITLE",
        nullable=false)
    private String title;
    private CoverType coverType;
    private Date publicationDate;
    @Transient
    private BigDecimal discount;
}
```

Entities and relationships

- If entities contained only simple persistent state, the issue of ORM would be a trivial one
- In fact, most entities need to be able to have **relationships** with other entities
- NOTE: there is an ambiguity in the meaning of “relationship” in the Entity-Relationship conceptual model and in the Object Model
- From now on we will use relationship in the sense of the Object Model



Relationship concepts

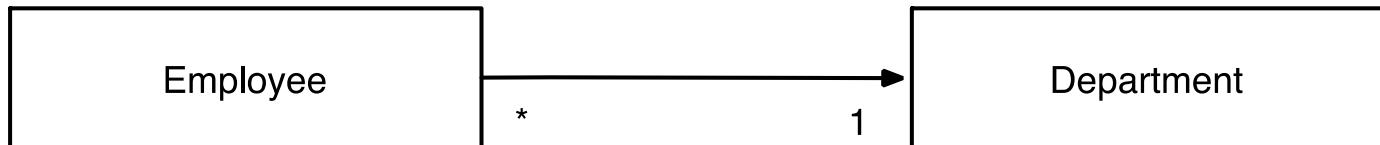
- Every (OM) relationship has four characteristics:
 - **Directionality**: each of the two entities may have an attribute that enables access to the other one
 - **Role**: each entity in the relationship is said to play a role with respect to one direction of access
 - **Cardinality**: the number of entity instances that exist on each side of the relationship
 - **Ownership**: one of the two entity in the relationship is said to own the relationship

Directionality

- Each entity in the relationship may have a **reference** to the other entity:
 - When each entity refers to the other, the relationship is **bidirectional**
 - If only one entity has a reference to the other, the relationship is said to be **unidirectional**
 - The reference can be single-valued or set-valued (collection of references)
- All relationships in JPA are unidirectional
 - A bidirectional relationship is intended as a “matched” pair of unidirectional mappings
 - “Matching” must be declared explicitly (via the dedicated attribute `mappedBy`)

Roles

- Based on directionality, one entity plays the role of **source** and one the role of **target**
- In the relationship from Employee to Department
 - Employee **is the source entity**
 - Department **is the target entity**

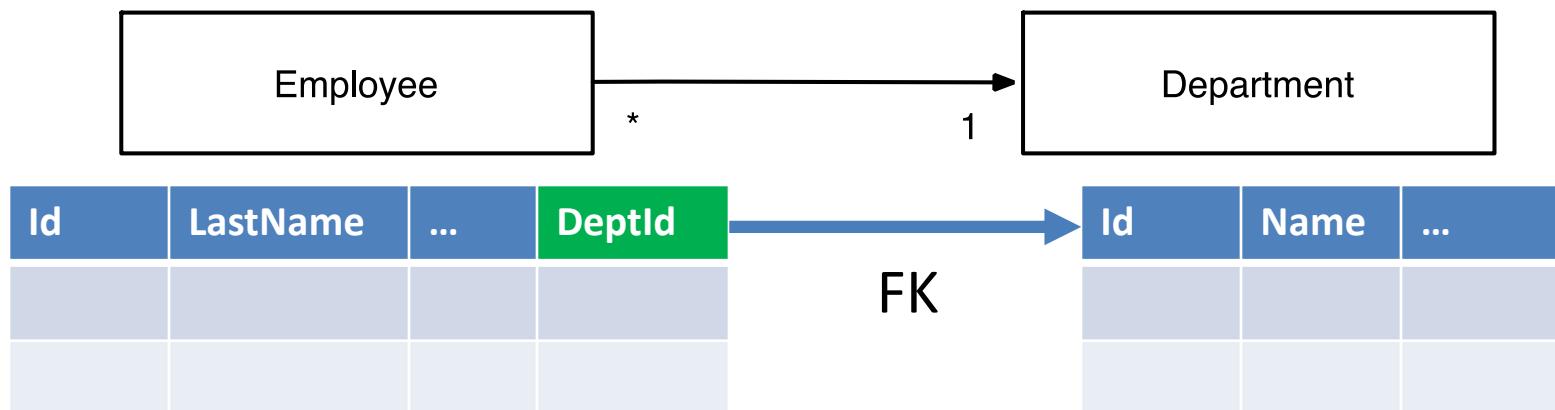


Cardinality

- Each role in the relationship has its own cardinality. This leads to four possible combinations:
 - **Many-to-one**: many source entities, one target entity
 - **One-to-many**: one source entity, many target entities
 - **One-to-one**: one source entity, one target entity
 - **Many-to-many**: many source entities, many target entities
- Remember: bidirectional relationships (the E-R way) are just pairs of matched unidirectional relationships with swapped source and target entities
- A **to-one** relationship implies that the source entity has **one reference** to the target entity,
- A **to-many** relationship implies that the source entity has a **multiple references** (collection) to the target entity

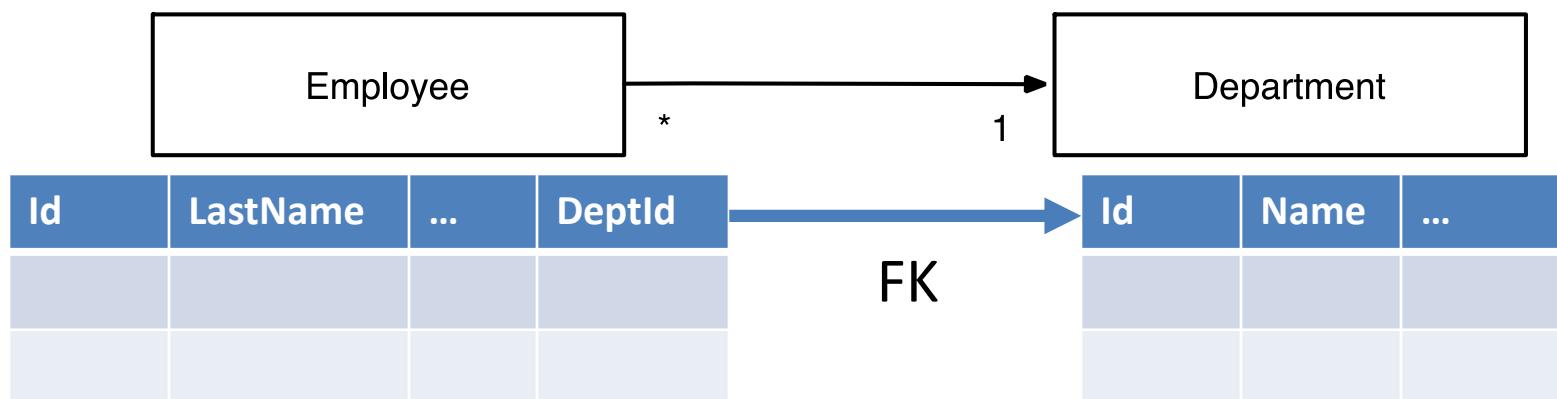
Ownership (1/2)

- In the database, relationships are implemented by a foreign key column that refers to the key of the referenced table
 - In JPA, such a column is called **join column**
- Example:
 - Many-to-one unidirectional relationship between Employee and Department
 - The underlying Employee table has a FK column containing the Department primary key



Ownership (2/2)

- In 1:N and 1:1 relationships, one of the two entities will have the FK column in its table
 - This entity is called the **owner** of the relationship and its side is called the **owning side**
- Example: Employee is the owner of the relationship
- Ownership is important because the annotations that define the physical mapping of the relationship (FK column) are specified in the owner side of the relationship



Possible relationship mappings

- 1:N
 - Bidirectional
 - Unidirectional one-to-many & many-to-one
- 1:1
 - Bidirectional
 - Unidirectional
- N:M
 - Bidirectional
 - Unidirectional
- Whether to go for uni- or bi-directionality depends on the application
- If the relationship is accessed from one entity to the other and vice versa, use bi-directionality otherwise use uni-directionality
- NOTES:
 - Bi-directionality can be also implemented with unidirectional relationship mapping + set-valued query
 - Performance considerations may prompt for the definition of bidirectional mappings even if only one direction is used

1:N bidirectional

- Expressed with (@ManyToOne + @OneToMany + mappedBy)
- The many-to-one mapping direction is defined by annotating the entity that participates with multiple instances with the @ManyToOne annotation
- The entity that contains the @ManyToOne annotation is the **owner** of the relationship because its underlying table stores the FK to the referenced table
- The @JoinColumn annotation can be used to specify the FK column of underlying table (defaults to the data member name)

@ManyToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    @JoinColumn(name="dept_fk")
    private Department dept;
    ...
}
```

1:N bidirectional, continued

- To achieve bi-directionality, the one-to-many mapping direction must be specified too
- This is done by including a `@OneToMany` annotation in the entity that participates with one instance
- The `@OneToMany` annotation is placed on a **collection** data member and comprises a `mappedBy` element to indicate the property that implements the inverse of the relationship
- NOTE: In a one-to-many mapping the owner of the relationship is still the entity that participates with multiple instances

`@OneToMany` annotation in `Department.java`

```
@Entity
public class Department {
    @Id private int id;
    @OneToMany(mappedBy="dept")
    private Collection<Employee> employees;
    ...
}
```

The attribute on the target entity that implements the inverse of the relationship

Uni-directional 1:N: many-to-one

- Sometimes applications require the access to relationships only along one direction
- In this case the by-directional mapping is not necessary
- The many-to-one direction case is trivial
- Use just the same annotations as in the bidirectional case and skip the inverse mapping

```
@Entity  
public class Employee {  
    @Id private int id;  
  
    @ManyToOne  
    @JoinColumn(name="dept_fk")  
    private Department department;  
    ...  
}
```

Uni-directional 1:N: one-to-many

- The one-to-many direction is less trivial
- Two alternatives are viable performance-wise
 - Map the relationship as in the bi-directional case and use only the one-to-many direction
 - Do not map the collection attribute in the entity that participates with one instance and use a query instead to retrieve the correlated instances, relying on the inverse (many-to-one) relationship direction mapping

```
List<Employee> emps =  
entityManager.createQuery(  
    "SELECT e " +  
    "FROM Employee e " +  
    "WHERE e.dept.id = :deptId",  
    Employee.class)  
.setParameter("deptId", "DEIB")  
.getResultList();
```

- See later the explanation of the JPQL query language

@joincolumn vs mappedBy

- The annotation `@JoinColumn` indicates the FK column that implements the relationship in the database; such annotation is normally inserted in the entity owner of the relationship (i.e., the one mapped to the table that actually contains the FK column)
 - Used to drive the generation of the SQL code to extract the correlated instances
- The `mappedBy` attribute indicates that “this side” is the inverse of the relationship, and the owner resides in the "other" related entity
 - Used to specify bidirectional relationships

Why specifying MappedBy?

- In absence of the mappedBy parameter the default JPA mapping (created when the database is generated from the JPA entities) uses a bridge table (as for N:M relationships)
- The purpose of the mappedBy parameter is to instruct **JPA** NOT to create a bridge table as the relationship is already being **mapped by** a FK in the opposite entity of the relationship

One-to-one mapping (1/2)

- In a one-to-one mapping the owner can be either entity, depending on the database design
- A one-to-one mapping is defined by annotating the owner entity (the one with the FK column) with the `@OneToOne` annotation

`@OneToOne annotation in Employee.java`

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne
    private ParkingSpace parkingSpace;
    ...
}
```

One-to-one mapping (2/2)

- If the one-to-one mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the other non-owner entity, the `@OneToOne` annotations must come with the `mappedBy` element
- The presence of `mappedBy` tells JPA that the foreign key constraint is in the table mapping the OTHER entity

```
@OneToOne annotation (inverse side) in  
ParkingSpace.java
```

```
@Entity  
public class ParkingSpace {  
    @Id private int id;  
    @OneToOne(mappedBy="parkingSpace")  
    private Employee employee;  
    ...  
}
```

Many-to-many mappings (1/2)

- In a many-to-many mapping there is no FK column
 - Such a mapping is implemented by means of a **join table** (aka **bridge table**)
- Therefore, we can arbitrarily specify as owner either entity

@ManyToMany annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    private Collection<Project> projects;
    ...
}
```

Many-to-many mappings (2/2)

- If the many-to-many mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the other entity, the @ManyToMany annotation must come with the mappedBy element

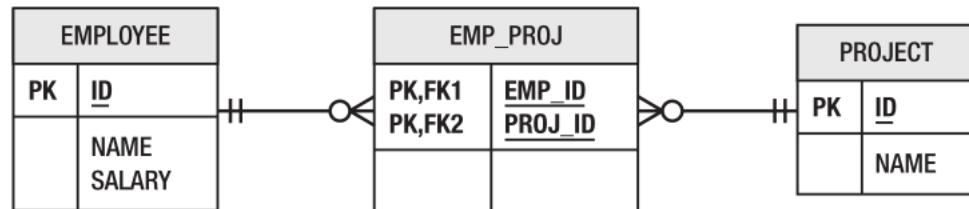
@ManyToMany annotation (inverse side) in Project.java

```
@Entity
public class Project {
    @Id private int id;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    ...
}
```

Join Table

- The logical model of a N:M relationships requires a bridge table (JOIN table in the JPA terminology)
- The non-default mapping of the entity to the bridge table is specified via annotation

```
@Entity  
public class Employee {  
    @Id private long id;  
    private String name;  
    @ManyToMany  
    @JoinTable(name="EMP_PROJ",  
        joinColumns=@JoinColumn(name="EMP_ID") ,  
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))  
    private Collection<Project> projects;  
    // ...  
}
```



- When no @JoinTable is specified, then a default name <Owner>_<Inverse> is assumed, where <Owner> is the name of the owning entity, and <Inverse> is the name of the inverse or non-owning entity (Employee_Project in the example)

Relationship fetch mode (1/3)

- **When loading** an entity, it is questionable if related entities are to be fetched & loaded too
 - Performance can be optimized by deferring data fetch until the time when they are needed
- This design pattern is called **lazy** (opposite= **eager**)
- Loading policy can be expressed specifying **fetch mode** for relationships (as seen for attributes of LOB type)
- At relationship level, lazy loading can greatly enhance performance because it can reduce the amount of SQL that is executed if correlated instances are seldom accessed by the application

Relationship fetch mode (2/3)

- When the **fetch mode** is not specified, by default:
 - A single-valued relationship is fetched **eagerly**
 - Collection-valued relationships are loaded **lazily**
- In case of bidirectional relationships, the fetch mode might be lazy on one side but eager on the other
 - Quite common situation, relationships are often accessed in different ways depending on the direction from which navigation occurs
- NOTE: Consider lazy loading as the most appropriate mode for all relationships, because if an entity has **many** single-valued relationships that are **not all used by applications**, the default eager mode may incur performance penalties

Lazy loading of the parkingSpace attribute

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    ...
}
```

Relationship fetch mode (3/3)

- The directive to lazily fetch an attribute is meant only to be a **hint** to the persistence provider
 - The provider is not required to respect the request because the behavior of the entity is not compromised if the provider decides to eagerly load
- The converse is **not true** because specifying that an attribute be eagerly fetched might be critical to access the entity once detached (i.e., no longer managed)

Cascading operations (1/4)

- By default, every Entity Manager operation applies **only to the entity supplied as an argument to the operation**
 - The operation **will not** cascade to other entities that have a relationship with the entity that is being operated on
- For some operations (e.g., `remove()`) this is usually the desired behavior

Cascading operations (2/4)

- Some other operations usually require cascading, such as `persist()`
 - If an entity has a relationship to another (dependent, aka “child”) entity normally the child entity must be persisted together with the father entity
- Example: Many-to-one unidirectional mapping between Employee and Address

Manually cascading

```
Employee emp = new Employee();  
Address addr = new Address();  
emp.setAddress(addr);  
em.persist(addr); ←  
em.persist(emp);
```

We would like to avoid explicit persisting the Address entity instance

Cascading operations (3/4)

- The cascade attribute is used to define when operations should be automatically cascaded across relationships
 - As the entity manager adds the Employee instance to the persistence context, it navigates the address relationship looking for a new Address entity to manage as well
 - The Address instance must be set on the Employee instance before invoking persist () on the employee object
 - If an Address instance has been set on the Employee instance and not persisted explicitly or implicitly via cascading, an error occurs

Enabling cascade persist

```
@Entity
public class Employee {
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
}
```

Cascading operations (4/4)

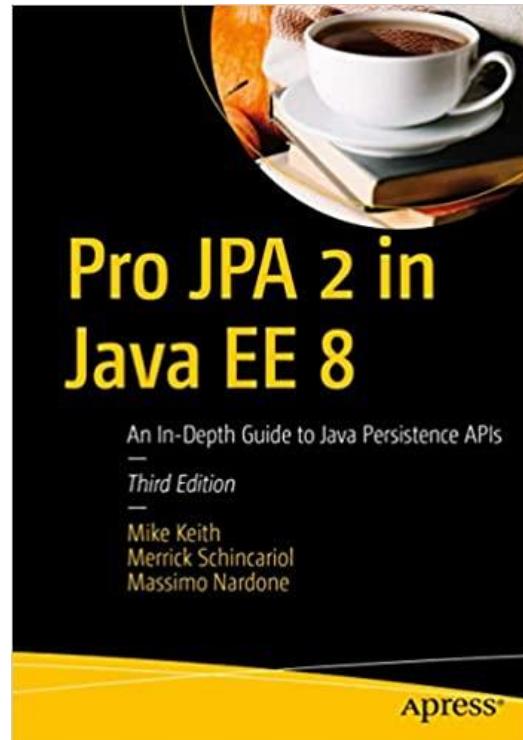
- The `cascade` attribute accepts several possible values specified in the `CascadeType` enumeration:
 - `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` and `DETACH`
 - `ALL` is a shorthand for declaring that all five operations should be cascaded
- As for relationships, cascade settings are unidirectional
 - they must be explicitly set on both sides of a relationship, if the default behavior has to be overridden
- NOTE: there is no **default cascade type in JPA**.
By **default** no operations are cascaded

Orphan removal

- JPA supports an additional remove cascading mode specified using `orphanRemoval` in `@OneToOne` and `@OneToMany`
 - appropriate for **privately owned** parent-child relationship, in which every child entity is associated only to one parent entity through just one relationship (weak entity in the E-R terminology)
 - causes the child entity to be removed when the parent-child relationship is broken either by removing the parent or by setting to null the attribute that holds the related entity, or in the one-to-many case, by removing the child entity from the parent's collection
- Difference between `CascadeType.REMOVE` and `orphanRemoval=true`
 - For orphan removal: invoking `setEmployees(null)` on Department the related Employee entities are removed from the database automatically
 - For remove cascade: invoking `setEmployees(null)` on Department, the related Employee entities are **NOT** removed from the database automatically

References

- JPA specifications,
[http://www.jcp.org/en/
jsr/detail?id=317](http://www.jcp.org/en/jsr/detail?id=317)
- Pro JPA 2 in Java EE8,
M. Keith, M. Schincariol,
Apress Media LLC
- *Chapter 4 and 10*



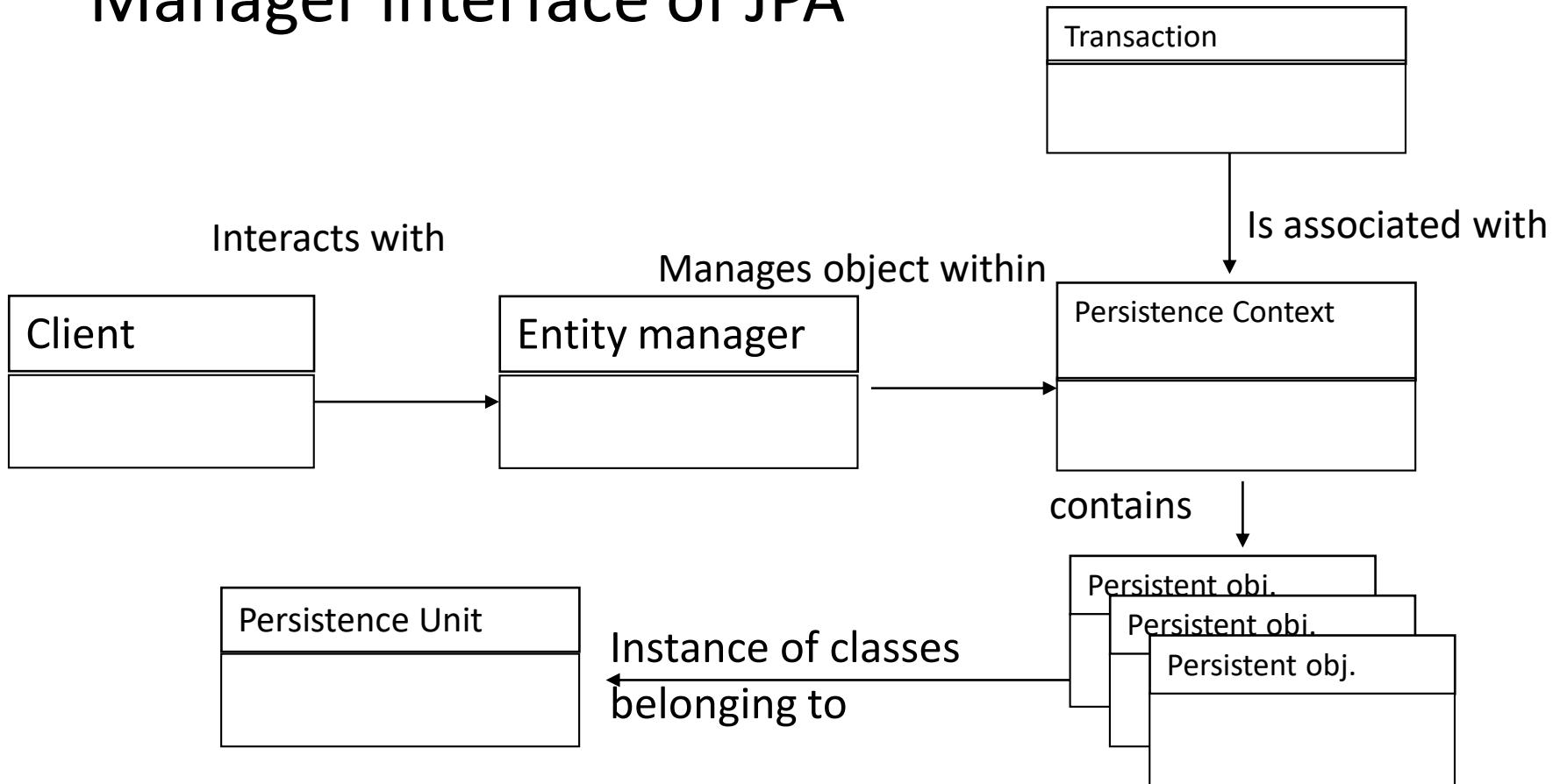
Java Persistence API

The Entity Manager

Databases 2

How to work with entities

- Entities are accessed through the Entity Manager interface of JPA



EntityManager

- Because entity instances are plain Java objects, they **do not become managed until the application invokes an API method to initiate the process**
- The Entity Manager is the central authority for all persistence actions
 - It manages the mapping between a fixed set of entity classes and an underlying data source
 - It provides APIs for creating queries, finding objects, and synchronizing objects and database state

Persistence Context

- It is the **fundamental (and somehow elusive) concept of JPA**: a kind of main memory database that holds the objects in the managed state
- A managed object is **tracked**, i.e., all the modifications to its state are monitored for **automatic alignment to the database**
- Database writes by default occur **asynchronously**, at a time decided by the persistence provider implementation
- For the database writes to happen, the **Persistence Context must hook up to a transaction**, which is the only means to write to the database
- A managed entity has **two lives**: one as a Java object and one as a relational tuple bound to it (ID of the POJO = PK of the tuple)
- Such a binding exists **ONLY inside the persistence context**. When the POJO exits the persistence context the binding breaks: it gets untracked and no longer synchronized to the database
- NOTE: the application **never sees the persistence context**, it interacts **ONLY** with the Entity Manager

EntityManager Interface

Method signature	Description
<code>public void persist(Object entity);</code>	Makes an entity instance become part of the persistence context, i.e., managed
<code>public <T> T find(Class<T> entityClass, Object primaryKey);</code>	Finds an entity instance by its primary key
<code>public void remove(Object entity);</code>	Removes an entity instance from the persistence context and thus from the database
<code>public void refresh(Object entity);</code>	Resets the state of entity instance from the content of the database
<code>public void flush();</code>	Writes the state of entities to the database as immediately as possible

Creating a new POJO

- Calling the `new` operator **does not** magically interact with some underlying service to create the `Employee` in the database
 - Instances of the `Employee` class remain POJOs until the application asks the `EntityManager` to start managing the entity
- When an entity is first instantiated, it is in the **transient** (or **new**) state since the `EntityManager` **does not know it exists yet**
- Transient entities are not part of the persistence context associated with the Entity Manager

Creating a new POJO

```
Employee emp = new Employee(ID, "John Doe"); //emp unknown to the EM!
```

Making an entity managed

- Entity Manager's `persist()` method makes a transient entity become **managed** (**BIG WARNING: managed <>> written to the DB!**)
- A managed entity “lives” in a persistence context and the EntityManager makes sure that any change to its state is tracked for being persisted to the database, **sooner or later**
- When the transaction associated with the Entity Manager's persistence context commits a new record is created in the database mapping the entity
- The Entity Manager `flush()` method can be called to ask the Persistence provider to write changes as soon as possible to the database
- The managed entity and the corresponding tuple become associated until the entity exits the managed state: changes to the entity will be reflected to changes to the tuple
- NOTE: calling `persist()` on an already managed entity instance is possible and triggers the cascade process

Persisting an entity

```
Employee emp = new Employee(ID, "John Doe");  
em.persist(emp); // emp now is managed
```

Finding an entity

- EntityManager's `find()` method takes as an input the class of the entity that is being sought and the primary key value that identifies the desired entity instance
- When the call completes, the returned object will be **managed**, added to the persistence context
 - If the entity instance is not found, then the `find()` method returns null
- The actual amount of data extracted and added to the persistence context depends on the **fetch policy** of attributes and relationships

Finding an entity

```
Employee emp = em.find(Employee.class, ID); //emp is managed
```

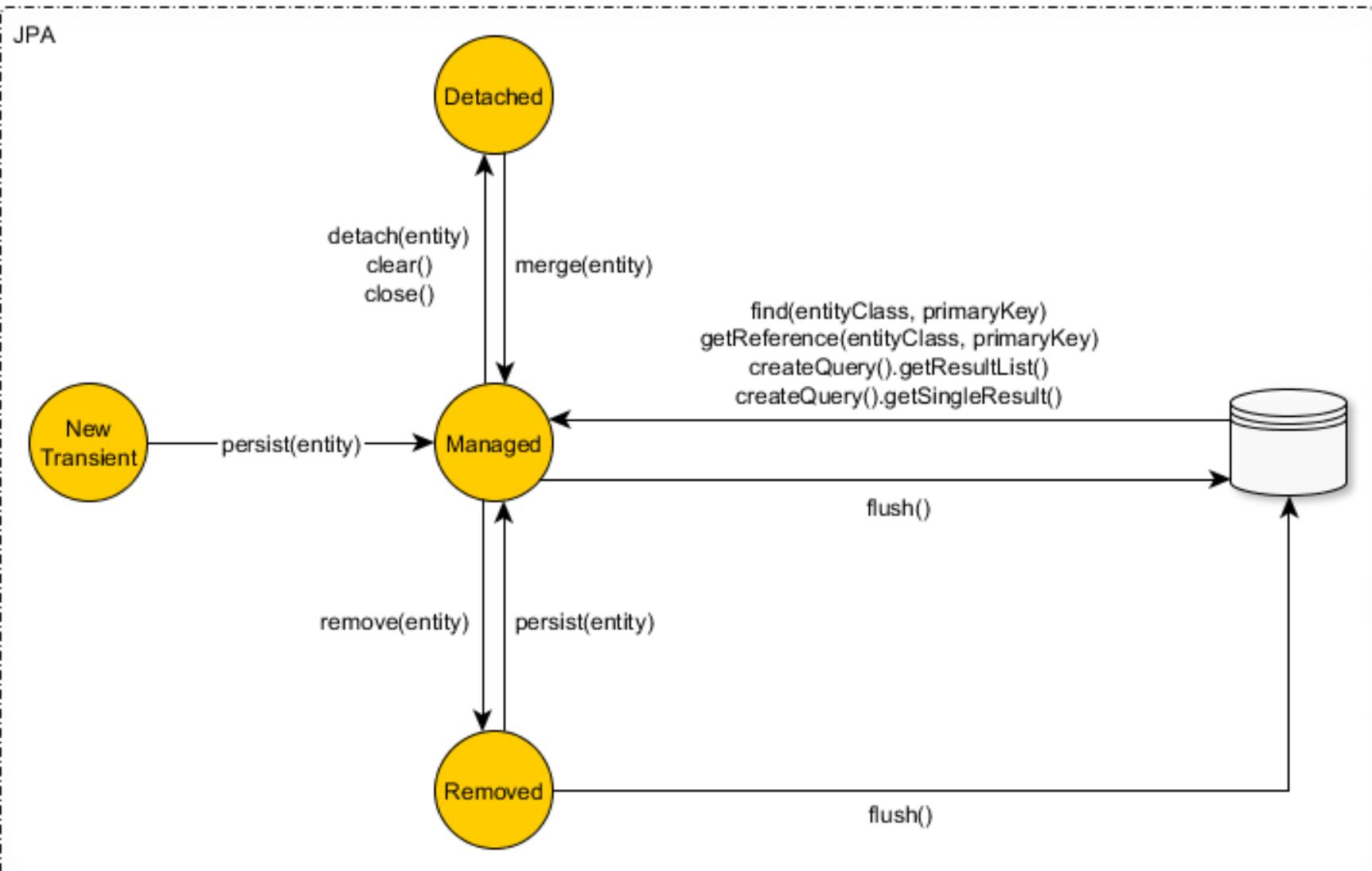
Removing an entity

- Entity Manager's `remove()` method breaks the association between the entity and the persistence context
- When the transaction associated with the Entity Manager's persistence context commits or the Entity Manager `flush()` method is called, the tuple associated with the entity is scheduled for deletion from the database
- The entity (Java object) still exists but its changes are no longer tracked for being (later) synchronized to the database

Removing an entity

```
em.remove(emp); // emp removed, no longer managed
```

EM operations & entity state transitions

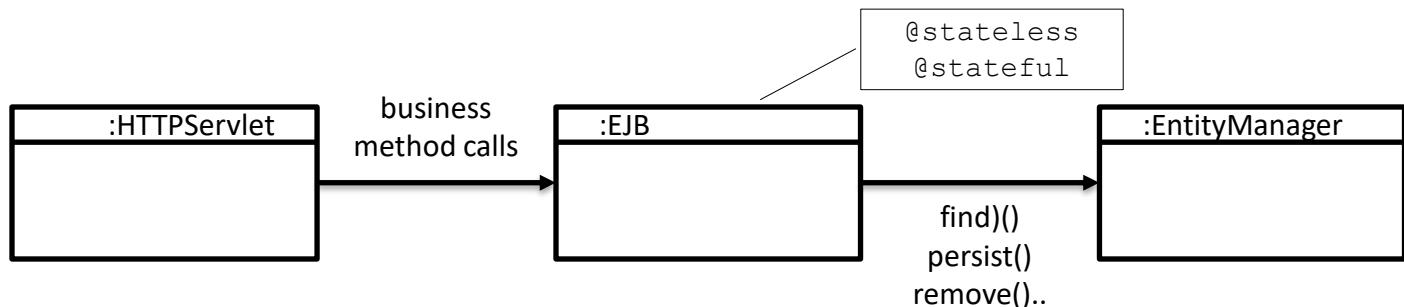


Entity lifecycle legenda

- **NEW**: Unknown to the Entity Manager, no persistent identity, no tuple associated
- **MANAGED**: Associated with persistence context, changes to objects automatically synch to database (**NOT VICE VERSA!**)
- **DETACHED**: Has an identity potentially associated with a database tuple but changes are NOT automatically propagated to database
- **REMOVED**: Scheduled for removal from the database
- **DELETED**: erased from the database

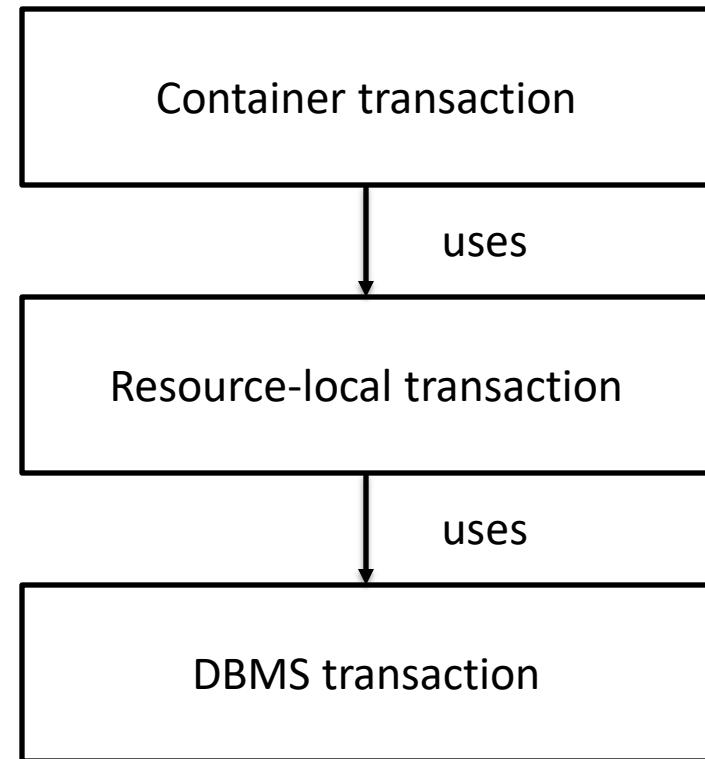
JPA application architectures

- JEE
 - The client exploits the services of a container (EJB or CDI) to connect to the Entity Manager
 - The client (e.g., a Web servlet) interacts with a business object (e.g., an EJB or a CDI component)
 - The business component interacts with the Entity Manager
 - The container provides the support to make JPA entity method calls **transactional** through the automatic creation of transactions



Transaction management in JPA

- Database application development requires understanding the transactional properties of code
- How the Entity Manager participates in transactions
- When a transaction starts/ends
- Even if transactions are managed transparently by the container, understanding how they work is necessary to predict application behaviour
- Transactions exist at three levels
 - DBMS transactions
 - Resource-local transactions
 - Container transaction



Transaction abstraction levels

- DBMS
 - They live inside the DBMS are demarcated by SQL commands and managed by the DBMS
- Resource-local
 - They are created and demarcated by means of JDBC commands issued through the JDBC connection interface. They are mapped by the JDBC driver to DBMS transactions.
They must be managed by the application
- Container (we will use this level)
 - They are defined through the JTA interface and mapped by the JTA Transaction Manager and EJB container to JDBC transactions. **They can be managed by the application or by the container (EJB, servlet or CDI)**

CM Entity Manager

- Container Managed

```
@Stateless // this is a business object (EJB)
public class myEJBService {
    @PersistenceContext(unitName = "MyPersistenceUnit")
    private EntityManager em;
```

- The container injects the EM instance into the business object
- The container creates and destroys instances of the Entity Manager transparently to the application
- This is the EM type **used by default**, if no other specification is added
- The container **provides the transaction** needed for saving the modifications made to the entities of the Persistence Context associated with the Entity Manager into the database

CM transaction: example

- (Web) client code (e.g., in a servlet)

```
@EJB(name = "it.polimi.db2.mission.services/MissionService")  
private MissionService mService; // client obtains a business object  
. . .  
Mission mission = mService.closeMission(missionId, userId);
```

- Business component code MissionService EJB

```
// transaction started here by the container  
public void closeMission(int missionId, int reporterId) throws  
    BadMissionReporter, BadMissionForClosing {  
  
    // persistence context created and associated with transaction  
    Mission mission = em.find(Mission.class, missionId);  
    if (mission.getReporter().getId() != reporterId) {  
        throw new BadMissionReporter(" . . .");  
    }  
    if (mission.getStatus() != MissionStatus.REPORTED) {  
        throw new BadMissionForClosing(" . . . ");  
    }  
    mission.setStatus(MissionStatus.CLOSED);  
  
} // transaction committed by container  
// no explicit transaction demarcation code needed!
```

Transaction and method calls

- When a client (e.g., a Web Servlet) calls a method of a business object that exploits a (default) CM EM for persistence, the (EJB) container provides (creates or reuses) a transaction for saving the modifications to the database
- If the called method in turns calls another method of the same or of a different business object, the same transaction is reused
- This is the most common and default behavior, but the business objects methods can be annotated to specify a different way to use the transactions provided by the container

Transactional behaviour of methods

- When **Container Managed** transactions are used the container wraps the method calls of managed components and execute them in a transaction
- A method can be annotated to obtain the desired transactional behaviour with @TransactionAttribute (TransactionAttributeType.XXX)
- Where XXX can be:
 - MANDATORY: a transaction is expected to have already been started and be active when the method is called. If no transaction is active, an exception is thrown
 - **REQUIRED: the default, if no transaction is active one is started. If one is active (e.g., created by the caller) this is used**
 - REQUIRES_NEW: the method always needs to be in its own transaction. Any active transaction is suspended
 - SUPPORTS: the method does not access transactional resources, but tolerates running inside one if it exists
 - NOT_SUPPORTED: the method will cause the container to suspend the current transaction if one is active when the method is called
 - NEVER: the method will cause the container to throw an exception if a transaction is active when the method is called.

Benefits of JPA: summary

- No need to write SQL code
- Application code ignores table names
 - Mapping is expressed via annotations with defaults
- No need to create/destroy the connection
- No need to create and terminate the transaction (begin, commit, rollback managed by the container)
- Business objects methods automatically made transactional (“all or nothing” semantics at the method call level and not at the SQL statement execution level)
- Default transactional behaviour, with annotations to specify different transaction management policies

Ranking Queries

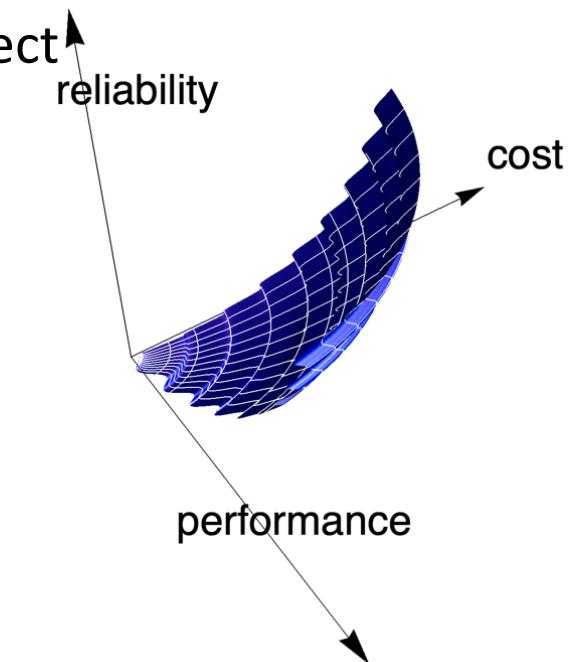
and other means
to find the best objects

Finding the best results

- Data are
 - available in large quantities
 - represented under many different models
 - queried with many different languages
 - processed with many different algorithms
- ...but how to get the best results out of the data?
- What does “best” even mean?

Multi-objective optimization

- Simultaneous optimization of different criteria
 - E.g., different attributes of objects in a dataset
- A general problem formulation:
 - Given N objects described by d attributes
 - and some notion of “goodness” of an object
 - Find the best k objects



Relevance of the problem

- Search engines, e-commerce
- Recommender Systems
- Machine Learning
 - k nearest neighbors
 - Feature selection
 - Classification
- Caching
- ...

Classical applications

- Multi-criteria queries
 - Example: ranking hotels by combining criteria about available facilities, driving distance, stars, ...

Search

Destination/property name:
Tokyo

Check-in date
19 Thursday 19 December 2...

Check-out date
20 Friday 20 December 2019

1-night stay

2 adults

No children ▾ 1 room ▾

I'm travelling for work ?

Search

Distance from Central Tokyo

<input type="checkbox"/> Less than 1 km	25
<input type="checkbox"/> Less than 3 km	272
<input type="checkbox"/> Less than 5 km	932

Online payment

<input type="checkbox"/> PayPal	1424
---------------------------------	------

Fun things to do

<input type="checkbox"/> Massage	328
<input type="checkbox"/> Hot tub/jacuzzi	133
<input type="checkbox"/> Bicycle rental (additional charge)	114
<input type="checkbox"/> Public Bath	113
<input type="checkbox"/> Sauna	86


Hotel Keihan Asakusa 3★ 
Taito, Tokyo · Show on map · 6 km from centre
Just booked for your dates 11 minutes ago
Black Friday Sale
Double Room with Small Double Bed - Non-Smoking
- 
1 double bed
Risk free: You can cancel later, so lock in this great price today.

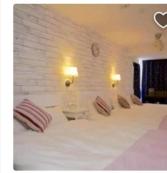
Very good 8.2
2,397 reviews

1 night, 2 adults
€ 58
Additional charges may apply
FREE cancellation
No prepayment needed
[Choose your room >](#)


Apartments Attract Kita Shinjuku 2★ 
Shinjuku Ward, Tokyo · Show on map · 4 km from centre
One-Bedroom Apartment - 
1 bedroom + 1 bathroom
1 single bed + 2 futon beds
20 m²
Only 1 left like this on our site

1 night, 2 adults
€ 120
includes taxes and charges

[See availability >](#)


Apartments Minato-Azabujuban 1 BR Apartment GAE52
Minato, Tokyo · Show on map · Metro access
 You missed it!
Your dates are popular – we've run out of rooms at this property! Check out more below.

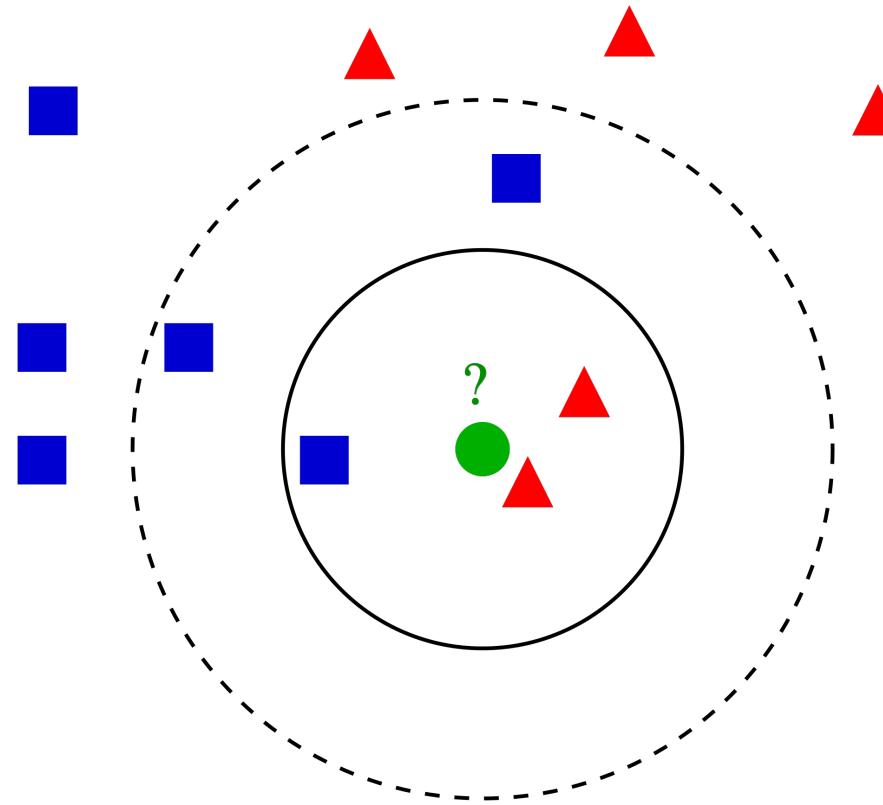
Good 7.8
9,677 reviews


Shinagawa Prince Hotel 4★ 
Minato, Tokyo · Show on map · 6 km from centre
Black Friday Sale
Twin Room - 
1 night, 2 adults
€ 140.21
includes taxes and charges

[See available room >](#)

Classical applications

- **k-Nearest Neighbors** (e.g., similarity search)
 - Given N points in some metric (d -dimensional) space, and a query point q in the same space, find the k points closest to q
 - Used for classification in Machine Learning



Multi-objective optimization

- Main approaches:
 - Ranking (aka top-k) queries
 - Top k objects according to a given scoring function
 - Skyline queries
 - Set of non-dominated objects

Historical perspective

Rank aggregation

[Borda, 1770][Marquis de Condorcet, 1785][Llull, 13th century]

- Rank aggregation is the problem of combining several ranked lists of objects in a robust way to produce a **single consensus ranking** of the objects

Rank aggregation

- Old problem (social choice theory) with lots of open challenges
- Given: n candidates, m voters

Candidate	Candidate	Candidate	Candidate	Candidate
A	B	D	E	C
B	D	B	A	E
C	E	E	C	A
D	A	C	D	B
E	C	A	B	D

Voter 1 Voter 2 Voter 3 Voter 4 Voter 5

- What is the overall ranking according to all the Voters?
 - No visible **score** assigned to candidates, only ranking
- Who is the best candidate?

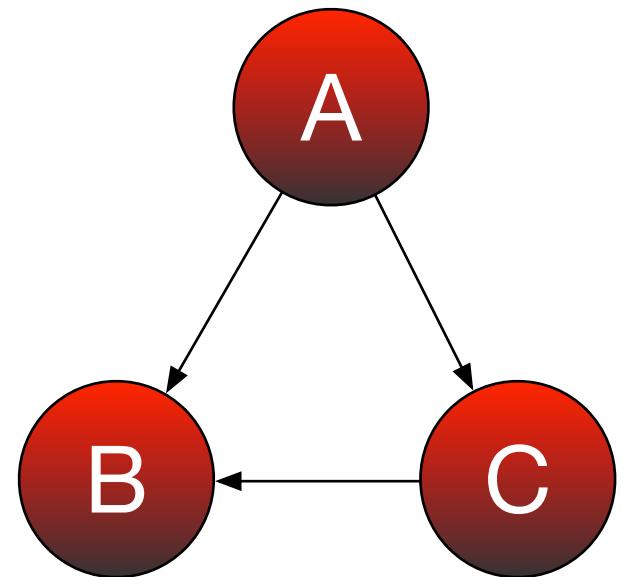
Borda's and Condorcet's proposals

- Borda's proposal
 - Election by order of merit
 - First place → 1 point
 - Second place → 2 points
 - ...
 - n-th place → n points
 - Candidate's penalty: sum of points
- **Borda** winner: candidate with the lowest penalty
- **Condorcet** winner:
 - A candidate who defeats every other candidate in pairwise majority rule election

Borda winner ≠ Condorcet winner

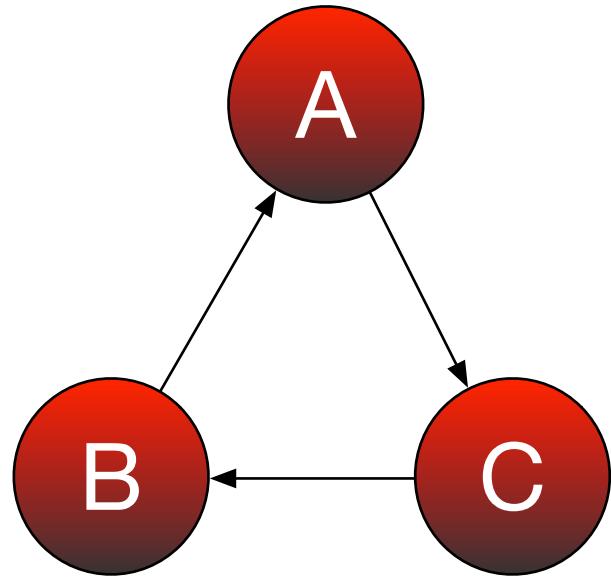
1	2	3	4	5	6	7	8	9	10
A	A	A	A	A	A	C	C	C	C
C	C	C	C	C	C	B	B	B	B
B	B	B	B	B	B	A	A	A	A

- Borda scores:
 - A: $1 \times 6 + 3 \times 4 = 18$
 - B: $3 \times 6 + 2 \times 4 = 26$
 - C: $2 \times 6 + 1 \times 4 = 16 \leftarrow \text{Borda winner}$
- Condorcet's criterion:
 - A beats both B and C in pairwise majority
 - A is Condorcet's winner



Condorcet's paradox

1	2	3
C	B	A
B	A	C
A	C	B



- Condorcet's winner may not exist
 - Cyclic preferences

Approaches to rank aggregation

[Arrow, 1950]

■ Axiomatic approach

- Desiderata of aggregation formulated as “axioms”
- Arrow’s result: a small set of natural requirements cannot be simultaneously achieved by any nontrivial aggregation function
- **Arrow’s paradox:** no rank-order electoral system can be designed that always satisfies these “fairness” criteria:
 - No dictatorship (nobody determines, alone, the group’s preference)
 - If all prefer X to Y, then the group prefers X to Y
 - If, for all voters, the preference between X and Y is unchanged, then the group preference between X and Y is unchanged



Perfect
democracy is
unattainable!

Approaches to rank aggregation

- Metric approach
 - Finding a new ranking R whose **total distance** to the initial rankings R_1, \dots, R_n is **minimized**
 - Several ways to define a distance between rankings, e.g.:
 - Kendall tau distance $K(R_1, R_2)$, defined as the number of exchanges in a bubble sort to convert R_1 to R_2
 - Spearman's footrule distance $F(R_1, R_2)$, which adds up the distance between the ranks of the same item in the two rankings
 - Finding an exact solution is
 - computationally hard for Kendall tau (NP-complete)
 - tractable for Spearman's footrule (PTIME)
 - These distances are related:
$$K(R_1, R_2) \leq F(R_1, R_2) \leq 2 K(R_1, R_2)$$
 - and $F(R_1, R_2)$ admits efficient approximations (e.g., **median ranking**)

Combining opaque rankings

Combining opaque rankings

[Fagin, Kuvar, Sivakumar, SIGMOD 2003]

- Techniques using only the **position** of the elements in the ranking (a.k.a. **ranked list**)
 - no other associated score
- We review **MedRank**
 - Based on the notion of **median**, it provides a(n approximation of) Footrule-optimal aggregation

Input: integer k , ranked lists R_1, \dots, R_m of N elements

Output: the top k elements according to median ranking

1. Use **sorted accesses** in each list, one element at a time, until there are k elements that occur in more than $m/2$ lists
2. These are the top k elements

MedRank example: hotels in Paris

- Three rankings
 - By price, by rating, and by distance
 - Looking for the top 3 hotels by “median rank”
- Strategy:
 - Make one sorted access at a time in each ranking
 - Look for hotels that appear in at least 2 rankings

NB: price, rating and distance are opaque, only the position matters

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	

Top 3 hotels	Median rank

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	

Top 3 hotels	Median rank

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	

Top 3 hotels	Median rank

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	



Top 3 hotels	Median rank
Novotel	median{2,3,?}=3

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	



Top 3 hotels	Median rank
Novotel	median{2,3,?}=3

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	



Top 3 hotels	Median rank
Novotel	median{2,3,5}=3
Hilton	median{4,5,?}=5
Ibis	median{1,5,?}=5

- The (maximum) number of sorted accesses made on each list is also called the **depth** reached by the algorithm
- Here, the depth is 5

price	rating	distance
Ibis	Crillon	Le Roch
Etap	Novotel	Lodge In
Novotel	Sheraton	Ritz
Mercure	Hilton	Lutetia
Hilton	Ibis	Novotel
Sheraton	Ritz	Sheraton
Crillon	Lutetia	Mercure
...	...	



Top 3 hotels	Median rank
Novotel	median{2,3,5}=3
Hilton	median{4,5,?}=5
Ibis	median{1,5,?}=5

When the median ranks are all distinct (unlike here), we have the Footrule-optimal aggregation

Optimality of MedRank

- An algorithm is **optimal** if its execution cost is never worse than any other algorithm on any input
- MedRank is **not optimal**
- However, it is **instance-optimal**
 - Among the algorithms that access the lists in sorted order, this is the **best possible algorithm** (up to a constant factor) on every input instance
 - In other words, its cost cannot be arbitrarily worse than any other algorithm on any problem instance

Definition of instance optimality

- A form of optimality aimed at when standard optimality is unachievable
 - Let \mathbf{A} be a family of algorithms, \mathbf{I} a set of problem instances
 - Let cost be a cost metric applied to an algorithm-instance pair
 - Algorithm A^* is **instance-optimal** wrt. \mathbf{A} and \mathbf{I} for the cost metric cost if there exist constants k_1 and k_2 such that, for all $A \in \mathbf{A}$ and $I \in \mathbf{I}$,
- $$\text{cost}(A^*, I) \leq k_1 \cdot \text{cost}(A, I) + k_2$$
- If A^* is instance-optimal, then any algorithm can improve wrt A^* by only a constant factor r , which is therefore called the **optimality ratio** of A^*
- Instance optimality is a much stronger notion than optimality in the average or worst case!
 - E.g., binary search is worst-case optimal, but not instance optimal
 - For each instance, a positive answer can be obtained in 1 probe, a negative answer in 2 probes (<< $\log N$)
- Example: MedRank
 - Optimality ratio = 2, additive constant = m
 - If we want to can get rid of dependence on m , we get optimality ratio = 4

Ranking queries
(a.k.a. top-k queries)

Top- k queries

Aim: to retrieve only the k best answers from a potentially (very) large result set

- Best = most important/interesting/relevant/...
- Useful in a variety of scenarios, such as e-commerce, scientific DB's, Web search, multimedia systems, etc.
- Needed: ability to “rank” objects (1st best, 2nd best, ...)

Ranking = ordering objects based on their “relevance”

Top-k queries: naïve approach (1)

- Assume a *scoring function* S that assigns to each tuple t a numerical score for ranking tuples
 - E.g. $S(t) = t.\text{Points} + t.\text{Rebounds}$
- Straightforward way to compute the result

Name	Points	Rebounds	...
Shaquille O'Neal	1669	760	...
Tracy McGrady	2003	484	...
Kobe Bryant	1819	392	...
Yao Ming	1465	669	...
Dwyane Wade	1854	397	...
Steve Nash	1165	249	...
...

Input: cardinality k , dataset R , scoring function S

Output: the k highest-scored tuples wrt S

1. for all tuples t in R
 - 1.1. compute $S(t)$
2. sort tuples based on their scores
3. return the first k highest-scored tuples

Top-k queries: naïve approach (2)

- Naïve approach expensive for large datasets
 - requires **sorting** a large amount of data
- Even worse if more than one relation is involved
$$S(t) = t.\text{Points} + t.\text{Rebounds}$$
 - Need to join all tuples, which is also costly
 - Here the join is **1-1**, but in general it can be **M-N** (each tuple can join with an arbitrary number of tuples)

Name	Points	...
Shaquille O'Neal	1669	...
Tracy McGrady	2003	...
Kobe Bryant	1819	...
Yao Ming	1465	...
Dwyane Wade	1854	...
Steve Nash	1165	...
...

Name	Rebounds	...
Shaquille O'Neal	760	...
Tracy McGrady	484	...
Kobe Bryant	392	...
Yao Ming	669	...
Dwyane Wade	397	...
Steve Nash	249	...
...

Top-k queries in SQL

- Two abilities required:
 - 1) **Ordering** the tuples according to their scores
 - 2) **Limiting** the output cardinality to k tuples
- Ordering is taken care of by ORDER BY
- Limiting was not native in SQL until 2008
 - FETCH FIRST k ROWS ONLY (SQL:2008)
 - Supported by IBM DB2, PostgreSQL, Oracle, MS SQL Server, ...
- Many non-standard syntaxes available:
 - ROWNUM <= k (ORACLE)
 - LIMIT k (PostgreSQL, MySQL)
 - SELECT TOP k FROM (MS SQL Server)

...

Shortcomings of ORDER BY

- Consider the following queries:

A) `SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
AND Price <= 21000
ORDER BY 0.8*Price+0.2*Miles`

B) `SELECT *
FROM UsedCarsTable
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price+0.2*Miles`

The values 0.8 and 0.2, also called **weights**, are a way to normalize our preferences on Price and Mileage

- Query A will likely miss some relevant answers (*near-miss*)
 - e.g., a car with a price of \$21,500 but very low mileage
- Query B will return all Audi/A4 in the dataset (*information overload*)

Semantics of top-k queries

- Only the first k tuples become part of the result
- If more than one set of k tuples satisfies the ORDER BY directive, any of such sets is a valid answer (non-deterministic semantics)

```
SELECT *
FROM R
ORDER BY Price
FETCH FIRST 3
ROWS ONLY
```

R

OBJ	Price
t15	50
t24	40
t26	30
t14	30
t21	40

OBJ	Price
t26	30
t14	30
t21	40

OBJ	Price
t26	30
t14	30
t24	40

Both are valid results

Top-k queries: examples

- The best NBA player (by points and rebounds):

```
SELECT *
FROM NBA
ORDER BY Points + Rebounds DESC
FETCH FIRST 1 ROW ONLY
```

- The 2 cheapest Chinese restaurants

```
SELECT *
FROM RESTAURANTS
WHERE Cuisine = 'Chinese'
ORDER BY Price
FETCH FIRST 2 ROWS ONLY
```

- The top-5 Audi/A4 (by price and mileage)

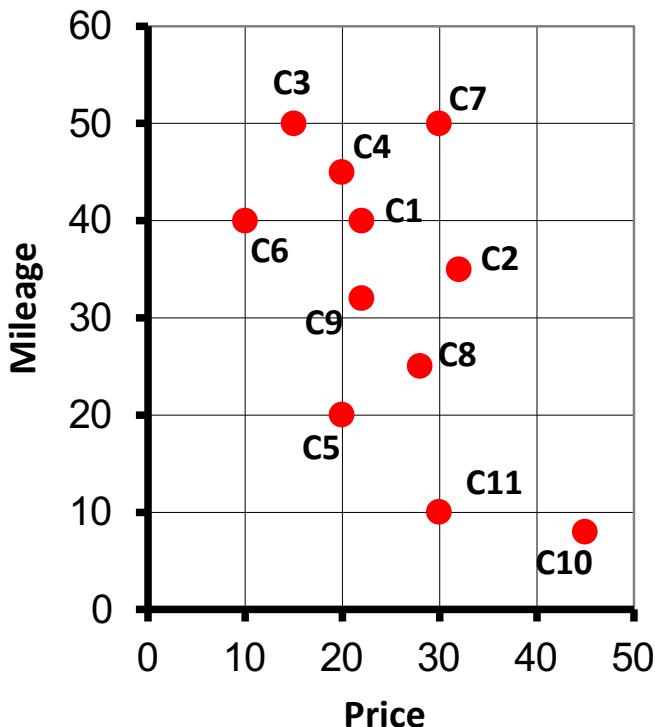
```
SELECT *
FROM USED CARS
WHERE Vehicle = 'Audi/A4'
ORDER BY 0.8*Price + 0.2*Miles
FETCH FIRST 5 ROWS ONLY
```

Evaluation of top-k queries

- Two basic aspects to consider:
 - query type: 1 relation, many relations, aggregate results, ...
 - access paths: no index, indexes on all/some ranking attributes
- Simplest case: top-k selection query with only 1 relation:
 - If input **sorted** according to S : read only first k tuples
 - No need to scan the entire input
 - Tuples **not sorted**: if k is not too large (which is the typical case), perform in-memory sort (through a *heap*)
 - The entire input needs to be read (cost $O(N \log k)$)

A geometric view

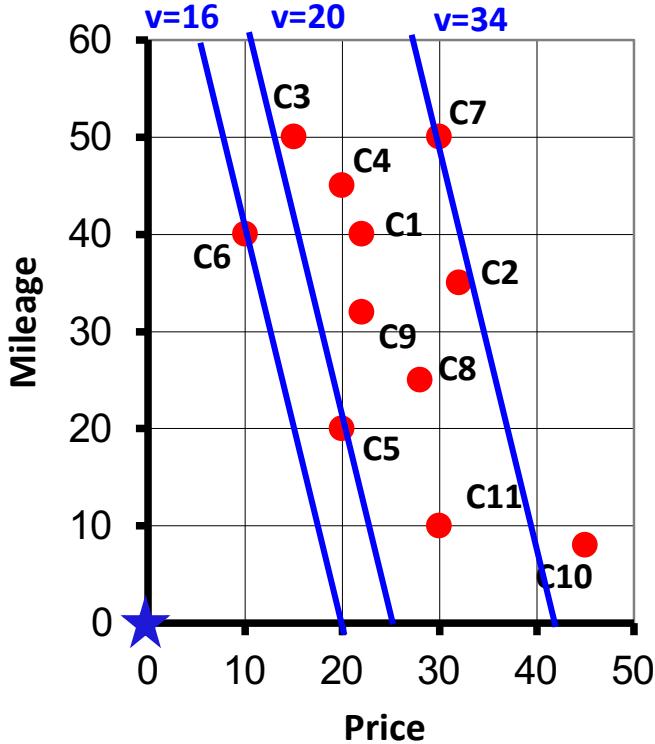
- Consider the 2-dimensional *attribute space* (Price,Mileage)



- Each tuple is represented by a 2-dimensional point (p,m) :
 - p is the Price value
 - m is the Mileage value
- Intuitively, minimizing $0.8 * \text{Price} + 0.2 * \text{Mileage}$ is equivalent to looking for points "close" to $(0,0)$
- $(0,0)$ is our (ideal) "target value" (i.e., a free car with 0 km's!)

The role of weights (preferences)

- Our preferences (e.g., 0.8 and 0.2) are essential to determine the result



- Consider the line of equation
$$0.8 \cdot \text{Price} + 0.2 \cdot \text{Mileage} = v$$
where v is a constant
- This can also be written as
$$\text{Mileage} = -4 \cdot \text{Price} + 5 \cdot v$$
from which we see that all the lines have a slope = -4
- By definition, all the points of the same line are “equally good”

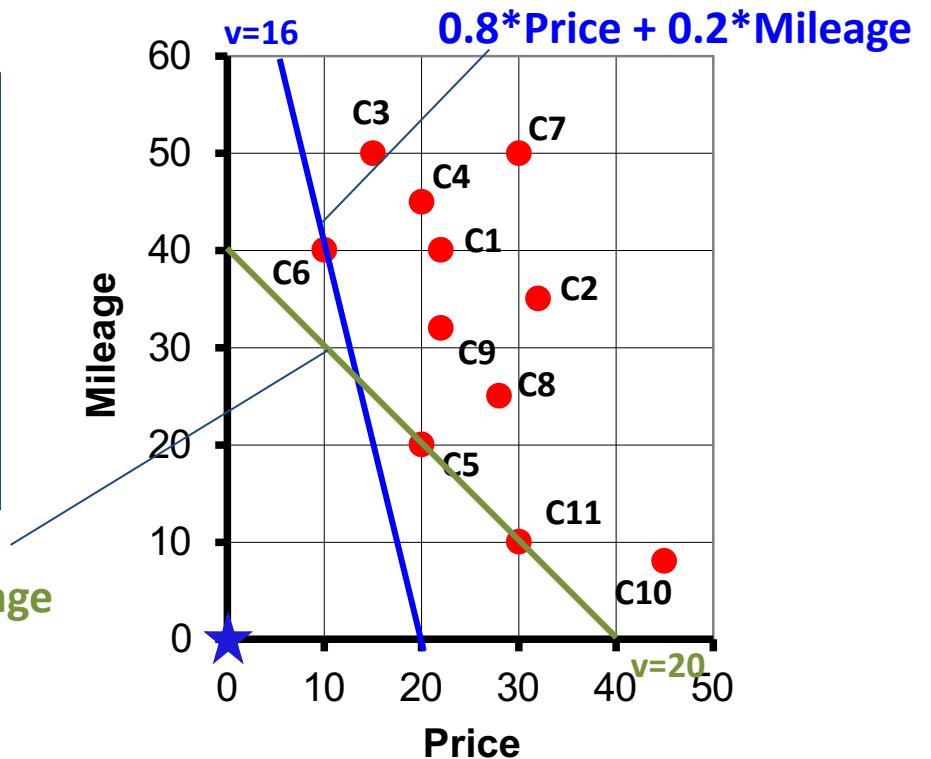
- With preferences $(0.8, 0.2)$ the best car is C6, then C5, etc.
- In general, preferences are a way to determine, given points (p_1, m_1) and (p_2, m_2) , which of them is “closer” to the target point $(0,0)$

Changing the weights

- Changing the weight values will likely lead to a different result

- With
 $0.8 * \text{Price} + 0.2 * \text{Mileage}$
the best car is C6
- With
 $0.5 * \text{Price} + 0.5 * \text{Mileage}$
the best cars are C5 and C11

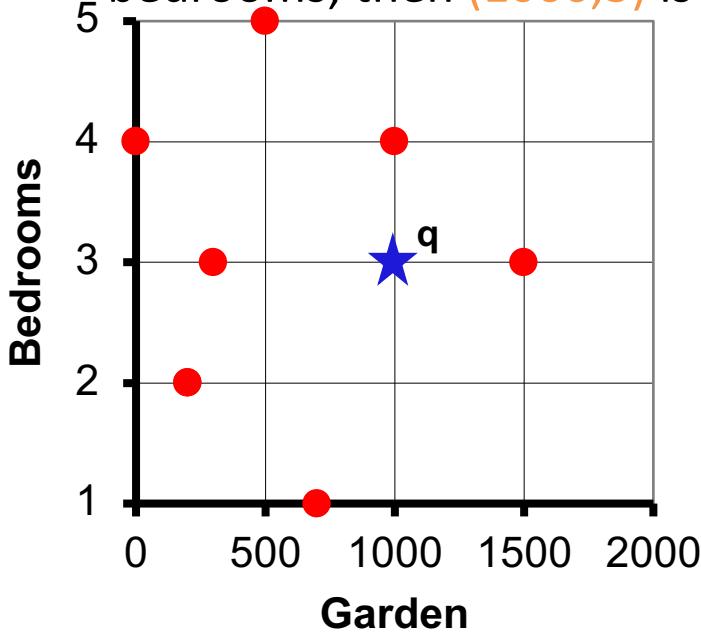
$0.5 * \text{Price} + 0.5 * \text{Mileage}$



- On the other hand, if weights do not change too much, the results of two top-k queries will likely have a high degree of overlap

Changing the target

- The target of a query is not necessarily $(0,0)$, rather it can be any point $q=(q_1, q_2)$ (q_i = query value for the i -th attribute)
- Example: assume you are looking for a house with a 1000 m^2 garden and 3 bedrooms; then $(1000, 3)$ is the target for your query



■ In general, in order to determine the “goodness” of a tuple t , we compute its “distance” from the target point q :
The lower the distance from q , the better t is

Note that distance values can always be converted into goodness “scores”, so that a higher score means a better match
Just change the sign and possibly add a constant,...

Top-k tuples = k-nearest neighbors

- It is sometimes useful to consider **distances** rather than scores
 - since most indexes are “distance-based”
- Therefore, the model is now:
 - An m -dimensional ($m \geq 1$) space $\mathbf{A} = (A_1, A_2, \dots, A_m)$ of ranking attributes
 - A relation $R(A_1, A_2, \dots, A_m, B_1, B_2, \dots)$, where B_1, B_2, \dots are other attributes
 - A target (query) point $q = (q_1, q_2, \dots, q_m)$, $q \in \mathbf{A}$
 - A function $d: \mathbf{A} \times \mathbf{A} \rightarrow \mathbb{R}$, measuring the distance between points in \mathbf{A} (e.g., $d(t, q)$ is the distance between t and q)
- Under this model, a top-k query is transformed into a so-called

k-Nearest Neighbors (k-NN) Query

- Given a point q , a relation R , an integer $k \geq 1$, and a distance function d
- Determine the k tuples in R that are closest to q according to d

Some common distance functions

- The most commonly used distance functions are L_p-norms (Minkowski distance):

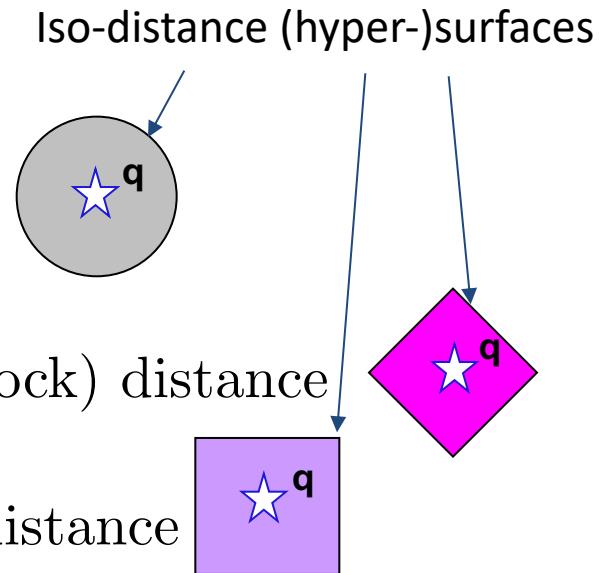
$$L_p(t, q) = \left(\sum_{i=1}^m |t_i - q_i|^p \right)^{1/p}$$

- Relevant cases are:

$$L_2(t, q) = \sqrt{\sum_{i=1}^m |t_i - q_i|^2} \text{ Euclidean distance}$$

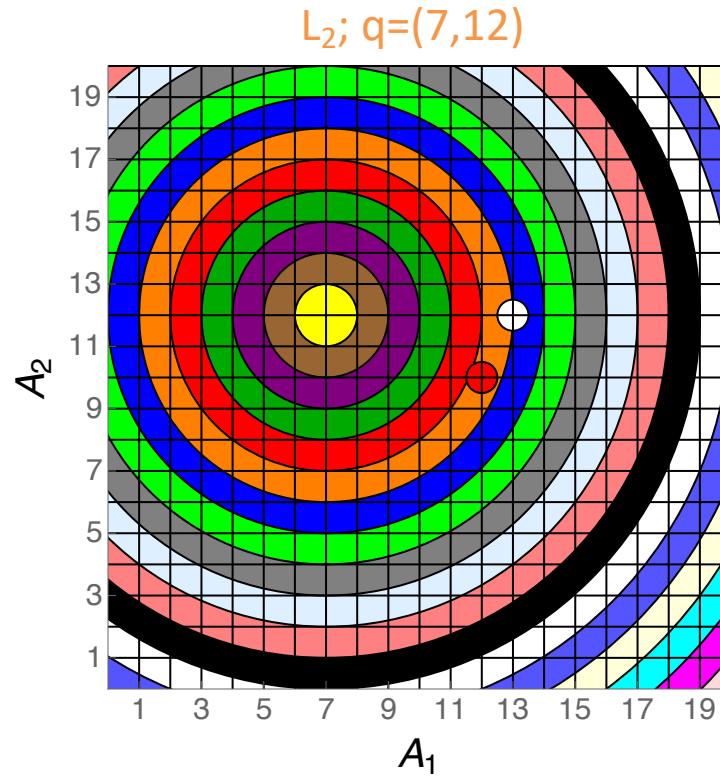
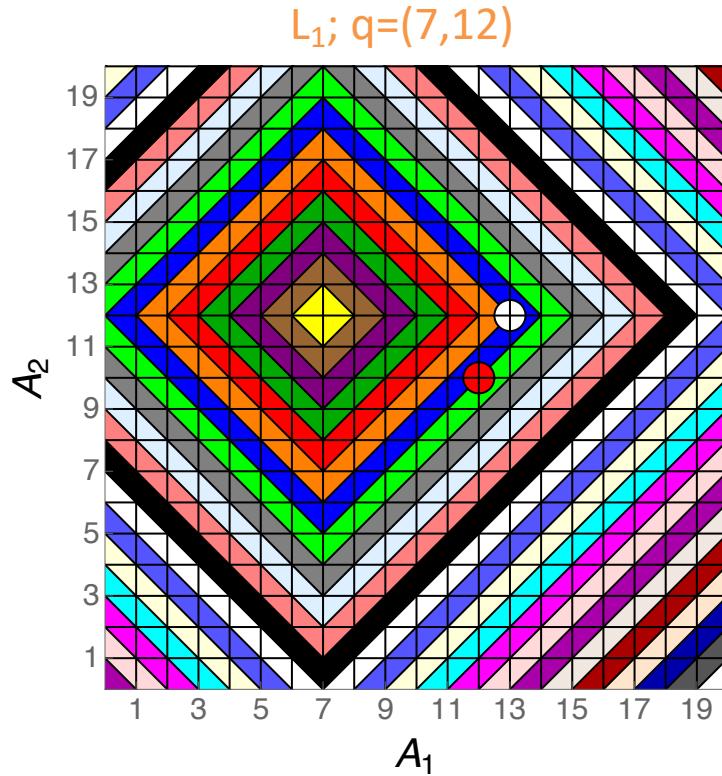
$$L_1(t, q) = \sum_{i=1}^m |t_i - q_i| \text{ Manhattan (city-block) distance}$$

$$L_\infty(t, q) = \max_i \{|t_i - q_i|\} \text{ Chebyshev (max) distance}$$



Shaping the attribute space

- Changing the distance changes the shape of the attribute space
 - each “stripe” corresponds to points with distance values between v and $v+1$, v integer

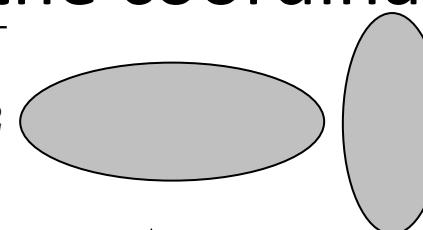


Note that, for 2 tuples t_1 and t_2 , it is possible to have
 $L_1(t_1, q) < L_1(t_2, q)$ and $L_2(t_2, q) < L_2(t_1, q)$

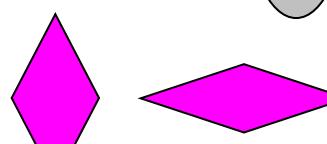
E.g.: $t_1=(13,12)$ ○
 $t_2=(12,10)$ ●

Distance functions with weights

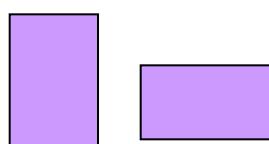
- The use of weights $W=(w_1, \dots, w_m)$ entails “stretching” some of the coordinates:

$$L_2(t, q; W) = \sqrt{\sum_{i=1}^m w_i |t_i - q_i|^2}$$


(hyper-)ellipsoids

$$L_1(t, q; W) = \sum_{i=1}^m w_i |t_i - q_i|$$


(hyper-)rhomboids

$$L_\infty(t, q; W) = \max_i \{w_i |t_i - q_i|\}$$


(hyper-)rectangles

- Thus, the scoring function

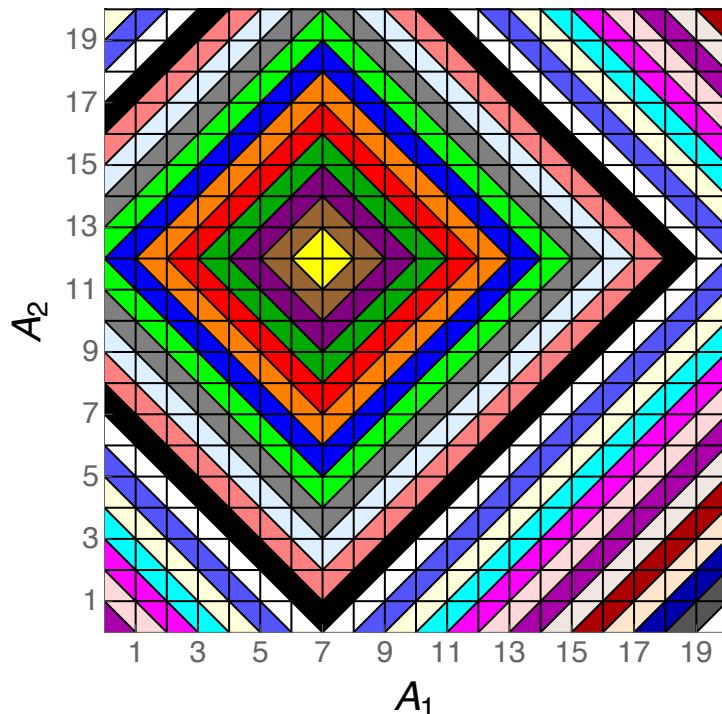
$0.8 * \text{Price} + 0.2 * \text{Mileage}$

is just a particular case of weighted L_1 distance

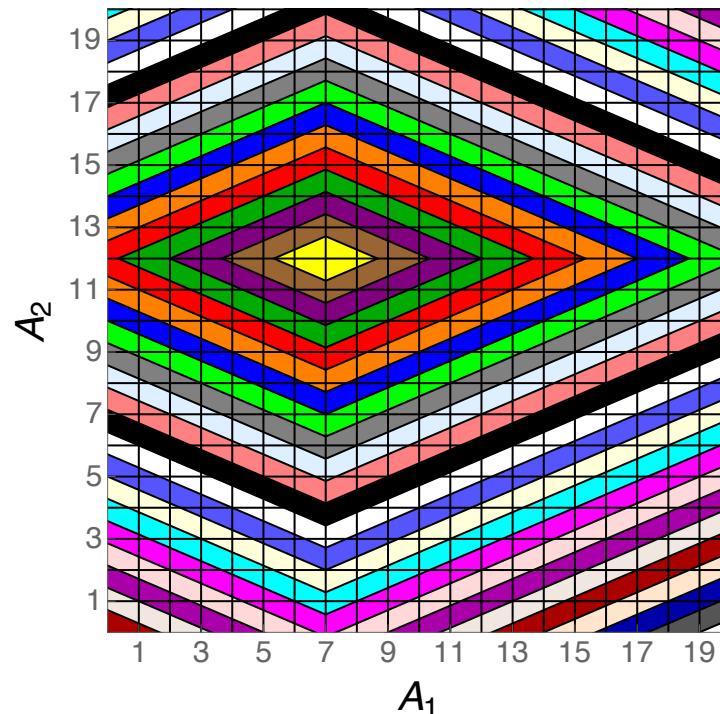
Weights the attribute space

- The figures show the effects of using L_1 with different weights

$L_1; q=(7,12) W=(1,1)$



$L_1; q=(7,12) W=(0.6,1.4)$



- If $w_2 > w_1$, then the hyper-rhombooids are more elongated along A_1
 - i.e., difference on A_1 values is less important than an equal difference on A_2 values

Top-k join queries

- In a **top-k join query** we have $n > 1$ input relations and a scoring function S defined on the result of the join, i.e.:

```
SELECT      <some attributes>
FROM        R1, R2, ..., Rn
WHERE       <join and local conditions>
ORDER BY    S(p1, p2, ..., pm) [DESC]
FETCH FIRST  k ROWS ONLY
```

where p_1, p_2, \dots, p_m are scoring criteria
– the “preferences”

Top-k join queries: examples

- The highest paid employee (compared to her dept budget):

```
SELECT E.*  
FROM EMP E, DEPT D  
WHERE E.DNO = D.DNO  
ORDER BY E.Salary / D.Budget DESC  
FETCH FIRST 1 ROW ONLY
```

- The 2 cheapest restaurant-hotel combinations in the same Italian city

```
SELECT *  
FROM RESTAURANTS R, HOTELS H  
WHERE R.City = H.City  
AND R.Nation = 'Italy' AND H.Nation = 'Italy'  
ORDER BY R.Price + H.Price  
FETCH FIRST 2 ROWS ONLY
```

Top-k 1-1 join queries

- All the joins are on a common key attribute(s)
 - Simplest case to deal with
 - Basis for the more general cases
 - Practically relevant
- Two main scenarios:
 - There is an index for retrieving tuples according to each preference
 - The relation is spread over several sites, each providing information only on part of the objects (the “middleware” scenario)

Top-k 1-1 join queries: assumptions

- Each input list supports **sorted access**:
 - Each access returns the id of the next best object, its partial score p_j (and possibly other attributes)
- Each input list supports **random access**:
 - Each access returns the partial score of an object, given its id (requires index on primary key)
- The id of an object is the same across all inputs (perhaps after reconciliation)
- Each input consists of the same set of objects

Top-k 1-1 join queries: example

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

- Aggregating restaurant reviews

```
SELECT *
FROM EatWell EW, BreadAndWine BW
WHERE EW.Name = BW.Name
ORDER BY EW.Score + BW.Score DESC
FETCH FIRST 1 ROW ONLY
```

Note: the winner is not the best locally!

Name	Global Score
Cheers!	16.8
The old mill	16.7
Da Gino	16.5
The canteen	16.0
Chez Paul	13.0
Let's eat!	12.4
Los pollos hermanos	11.5

A model for scoring functions

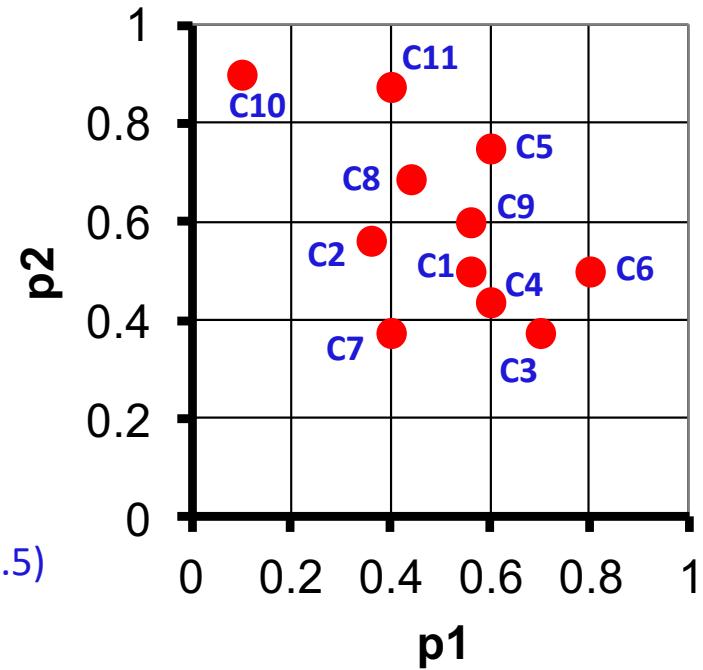
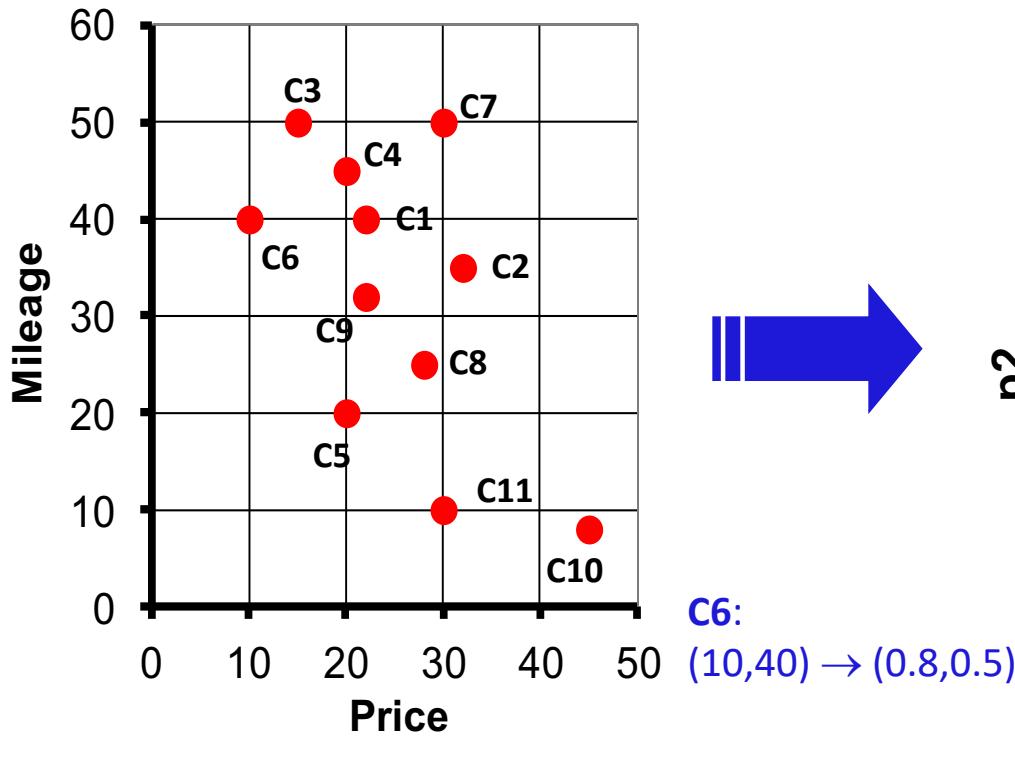
- Each object o returned by the input L_j has an associated local/partial score $p_j(o) \in [0,1]$
 - For convenience, scores are **normalized**
 - higher is better
 - only the best and the worst possible value of p_j matter
 - The hypercube $[0,1]^m$ is called the **score space**
- The point $p(o) = (p_1(o), p_2(o), \dots, p_m(o)) \in [0,1]^m$ is the map of o into the score space
- The **global score** $S(o)$ of o is computed by means of a **scoring function** S that combines in some way the local scores of o :

$$S : [0,1]^m \rightarrow \mathfrak{R}$$

$$S(o) \equiv S(p(o)) = S(p_1(o), p_2(o), \dots, p_m(o))$$

The score space

- Consider the attribute space $\mathbf{A} = (\text{Price}, \text{Mileage})$
- Let: $p_1(o) = 1 - o.\text{Price}/\text{MaxP}$, $p_2(o) = 1 - o.\text{Mileage}/\text{MaxM}$
- Let's take $\text{MaxP} = 50,000$ and $\text{MaxM} = 80,000$
- Objects in \mathbf{A} are mapped into the score space as shown
 - The relative order on each coordinate (local ranking) remains unchanged



Common scoring functions

SUM (AVG): used to weigh preferences equally

$$\text{SUM}(o) \equiv \text{SUM}(p(o)) = p_1(o) + p_2(o) + \dots + p_m(o)$$

WSUM (Weighted sum): to weigh the ranking attributes differently

$$\text{WSUM}(o) \equiv \text{WSUM}(p(o)) = w_1 * p_1(o) + w_2 * p_2(o) + \dots + w_m * p_m(o)$$

MIN (Minimum): just considers the worst partial score

$$\text{MIN}(o) \equiv \text{MIN}(p(o)) = \min\{p_1(o), p_2(o), \dots, p_m(o)\}$$

MAX (Maximum): just considers the best partial score

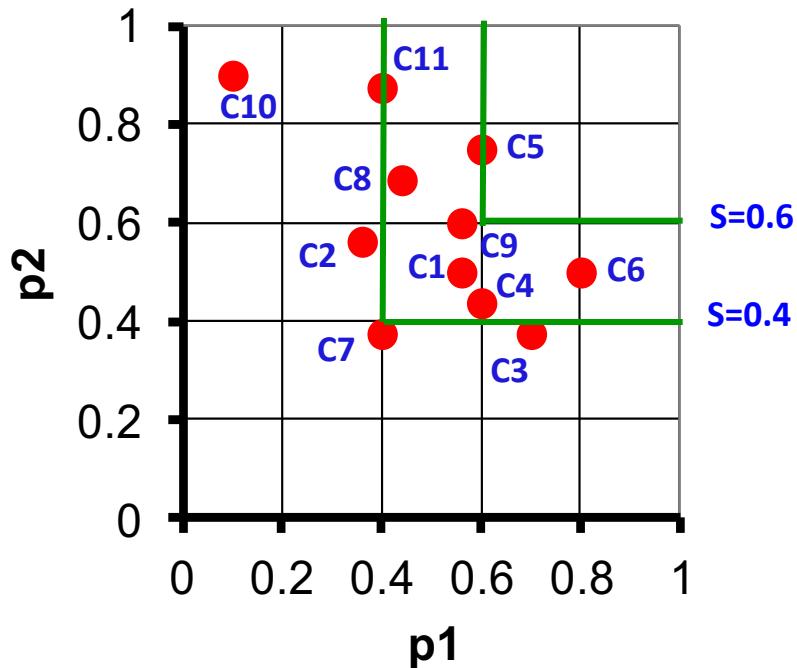
$$\text{MAX}(o) \equiv \text{MAX}(p(o)) = \max\{p_1(o), p_2(o), \dots, p_m(o)\}$$

(even with MIN) we always want to retrieve the k objects with the highest global scores

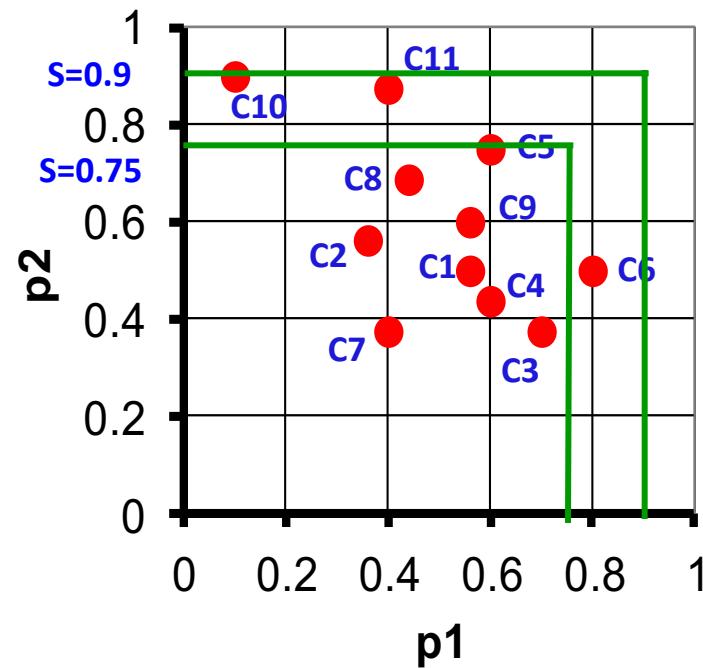
Equally scored objects

- We can define iso-score curves in the score space
 - Sets of points with the same global score
 - Similar to iso-distance curves in an attribute space

$$S(o) = \text{MIN}(p(o))$$



$$S(o) = \text{MAX}(p(o))$$



The simplest case: MAX

[Fagin, PODS 1996]

- We are now ready to ask “the big question”:

How can we efficiently compute the result of a top-k 1-1 join query using a scoring function S ?

- When $S \equiv \text{MAX}$, it is really simple:

You can use my algorithm B_0 ,
which just retrieves the best k objects from
each source, that's all!

Beware! B_0 only works for **MAX**.
Other scoring functions require
smarter, and more costly, algorithms



Ronald Fagin

The B_0 algorithm

Input: integer $k \geq 1$, ranked lists R_1, \dots, R_m

Output: the top- k objects according to the MAX scoring function

1. Make k sorted accesses on each list and store objects and partial scores in a buffer B
2. For each object in B , compute the MAX of its (available) partial scores
3. Return the k objects with maximum score

- No need to obtain missing partial scores
- No random accesses are executed

B_0 : examples

$k = 2$

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

B_0 : examples

$k = 2$

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

B_0 : examples

$k = 2$

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

B_0 : examples

$k = 2$

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

B_0 : examples

$k = 2$

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

Name	S
The old mill	9.2
Da Gino	9.0
The canteen	9.0
Cheers!	8.5

B_0 : examples

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	s
o7	1.0
o2	0.9
o3	0.65

k = 3

EatWell

Name	Score
The old mill	9.2
The canteen	9.0
Cheers!	8.3
Da Gino	7.5
Let's eat!	6.4
Chez Paul	5.5
Los pollos hermanos	5.0

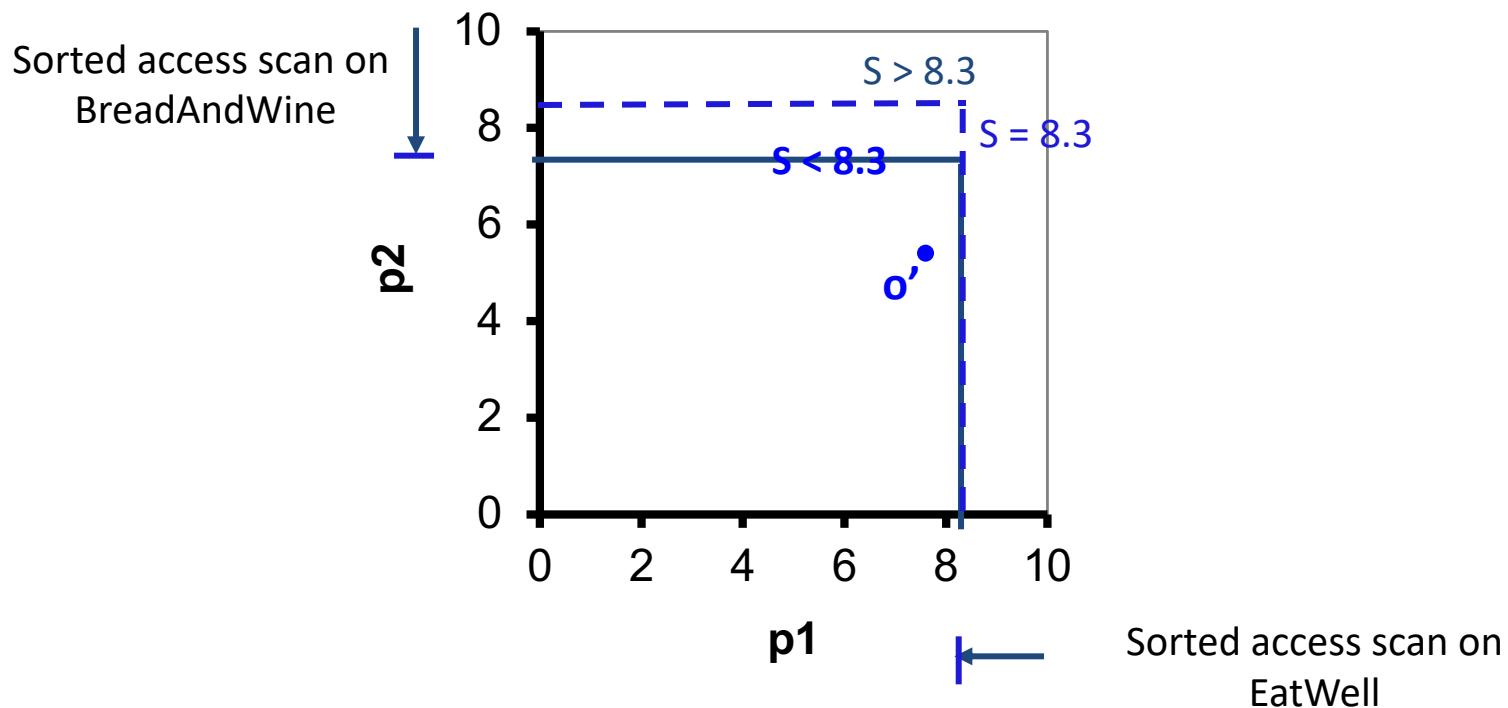
BreadAndWine

Name	Score
Da Gino	9.0
Cheers!	8.5
The old mill	7.5
Chez Paul	7.5
The canteen	7.0
Los pollos hermanos	6.5
Let's eat!	6.0

Name	S
The old mill	9.2
Da Gino	9.0
The canteen	9.0
Cheers!	8.5

Why B_0 works (restaurants example)

- After 3 sorted access rounds it is guaranteed that there are at least 3 restaurants o with $S(o) \geq 8.3$
- A restaurant like o' , that has not been retrieved by any sorted access scan cannot have a global score higher than 8.3



B_0 doesn't work when $S \neq \text{MAX}$

- Let $S = \text{MIN}$ and $k = 1$

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

B_0 doesn't work when $S \neq \text{MAX}$

- Let $S = \text{MIN}$ and $k = 1$

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

B_0 doesn't work when $S \neq \text{MAX}$

- Let $S = \text{MIN}$ and $k = 1$

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

OID	S
o2	0.95
o7	0.9

WRONG!!

B_0 doesn't work when $S \neq \text{MAX}$

- Let $S = \text{MIN}$ and $k = 1$

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

OID	S
o2	0.95
o7	0.9

WRONG!!

- What if we consider **all** the partial scores of the seen objects ($\{o2, o7\}$ in the figure)?
- After performing the necessary **random accesses** we get:

B_0 doesn't work when $S \neq \text{MAX}$

- Let $S = \text{MIN}$ and $k = 1$

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

OID	S
o2	0.95
o7	0.9

WRONG!!

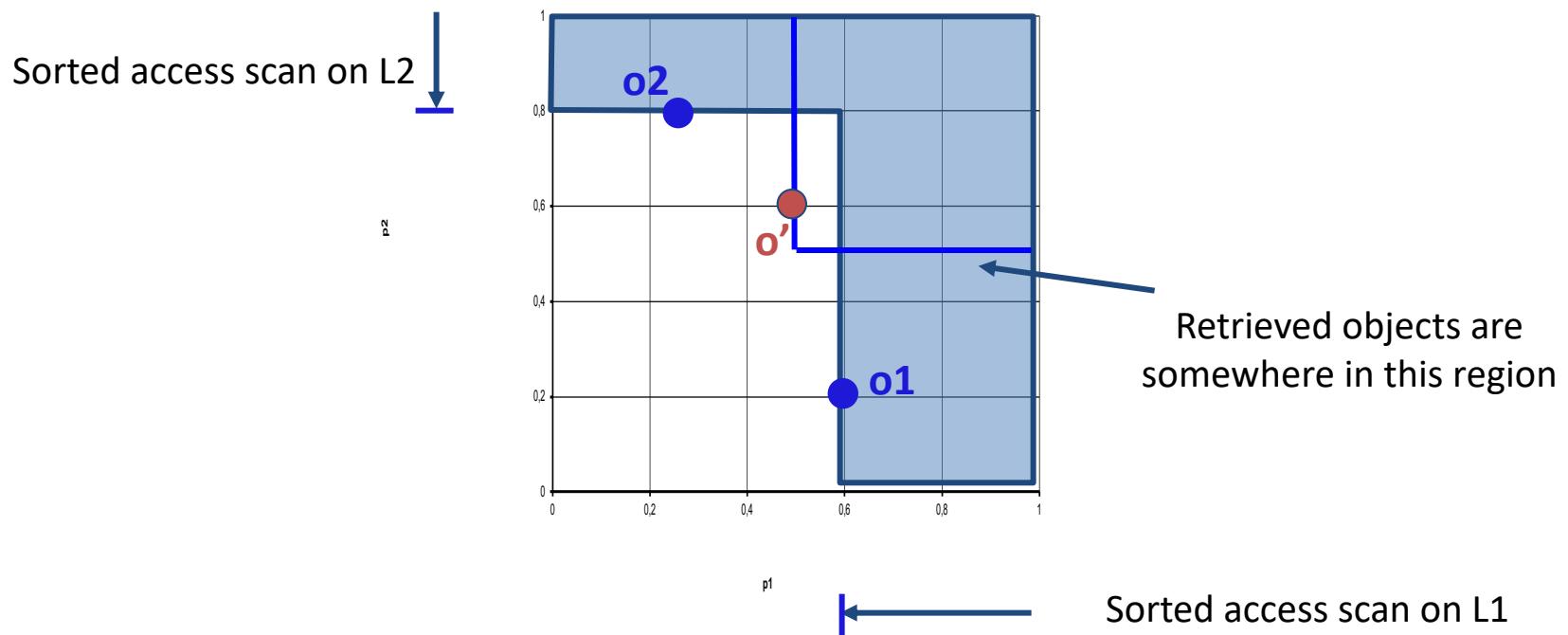
- What if we consider **all** the partial scores of the seen objects ($\{o2, o7\}$ in the figure)?
- After performing the necessary **random accesses** we get:

OID	S
o2	0.6
o7	0.5

STILL WRONG!!? 😞

Why B_0 doesn't work: graphical intuition

- Let $S \equiv \text{MIN}$ and $k = 1$
- When the sorted accesses terminate, we have no lower bound on the global scores of the retrieved objects (i.e., it might also be $S(o) = 0$)
- Any non-retrieved object, like o' , can now be the winner!
 - Note that, in this case, o' would be the best match even for $S \equiv \text{SUM}$



Fagin's Algorithm (FA)

[Fagin, PODS 1998]

Input: integer k , a **monotone** function S combining ranked lists R_1, \dots, R_m

Output: the top k \langle object, score \rangle pairs

1. Extract the same number of objects by **sorted accesses** in each list until there are at least k objects in common
2. For each extracted object, compute its overall score by making **random accesses** wherever needed
3. Among these, output the k objects with the best overall score

- Complexity is **sub-linear** in the number N of objects
 - Proportional to the square root of N when combining two lists (complexity is $O(N^{(m-1)/m}k^{1/m})$)
 - The stopping criterion is **independent** of the scoring function
 - Not instance-optimal

Example: hotels in Paris with FA

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

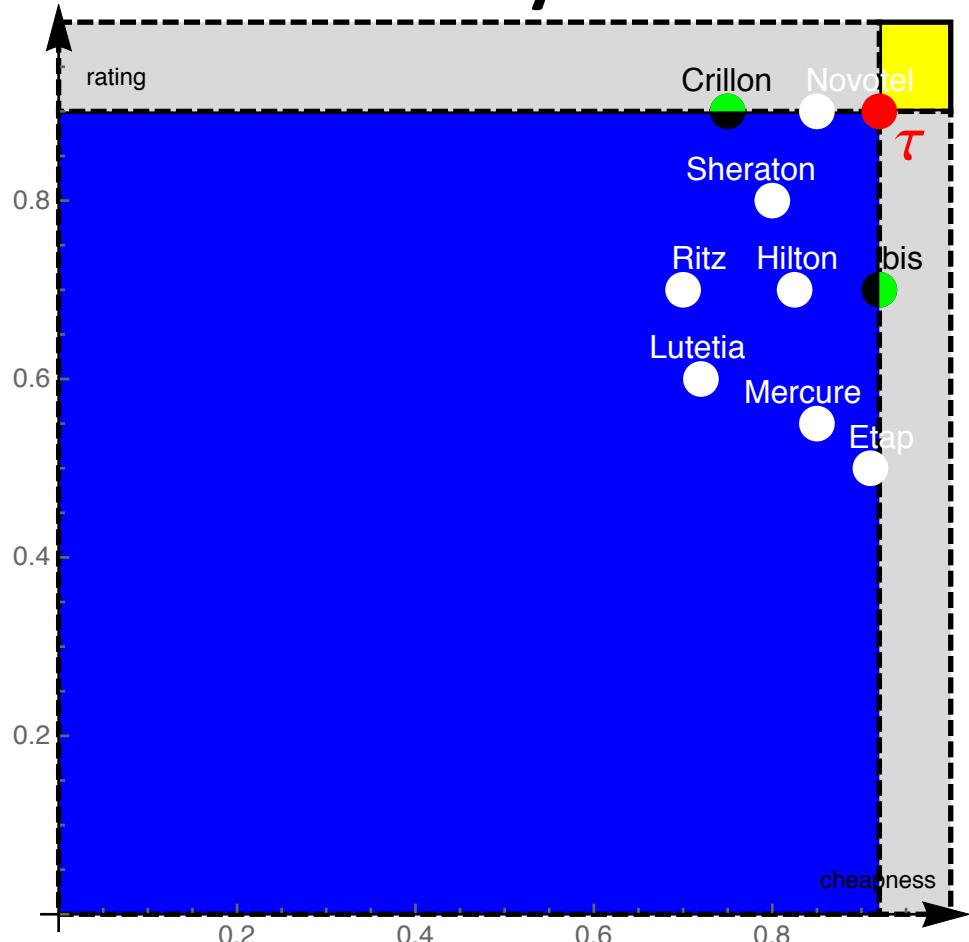
- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Look for hotels that appear in both lists

Hotels	Cheapness	Hotels	Rating
Ibis	.92	Crillon	.9
Etap	.91	Novotel	.9
Novotel	.85	Sheraton	.8
Mercure	.85	Hilton	.7
Hilton	.825	Ibis	.7
Sheraton	.8	Ritz	.7
Crillon	.75	Lutetia	.6
...		...	

Top 2	Score
Novotel	.875
Crillon	.825

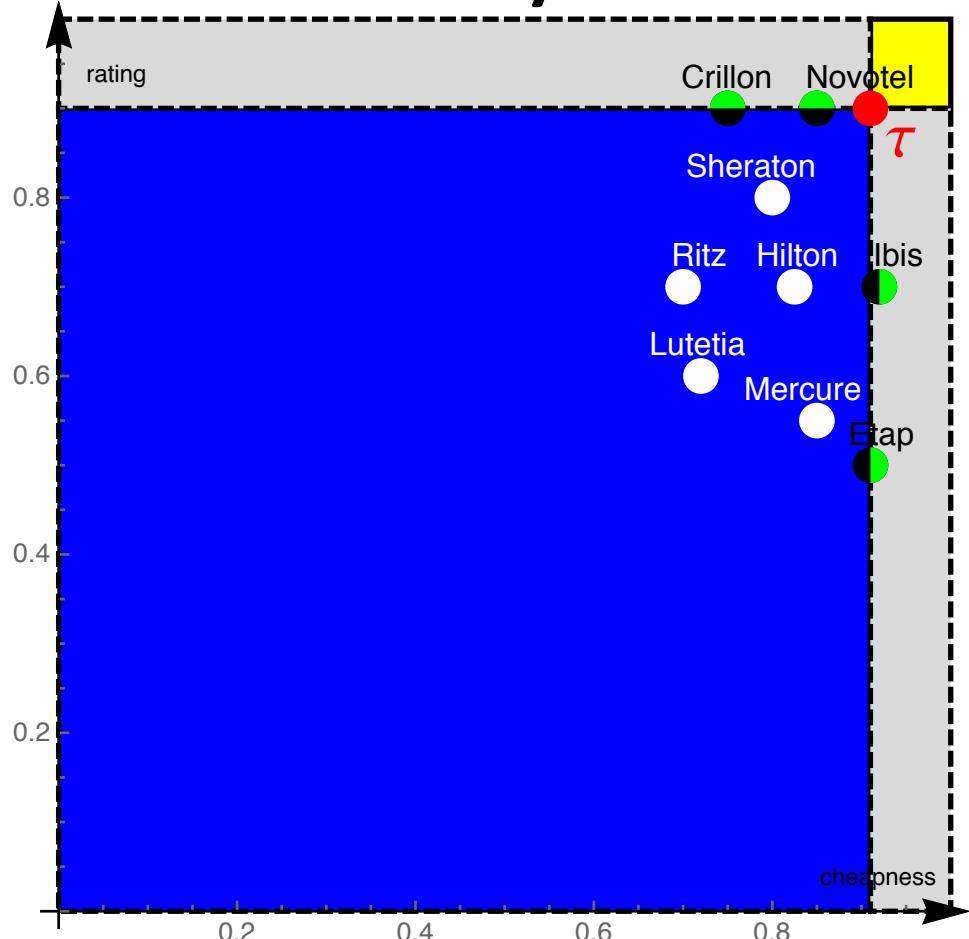
- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Now complete the score with random accesses

Why does FA work?



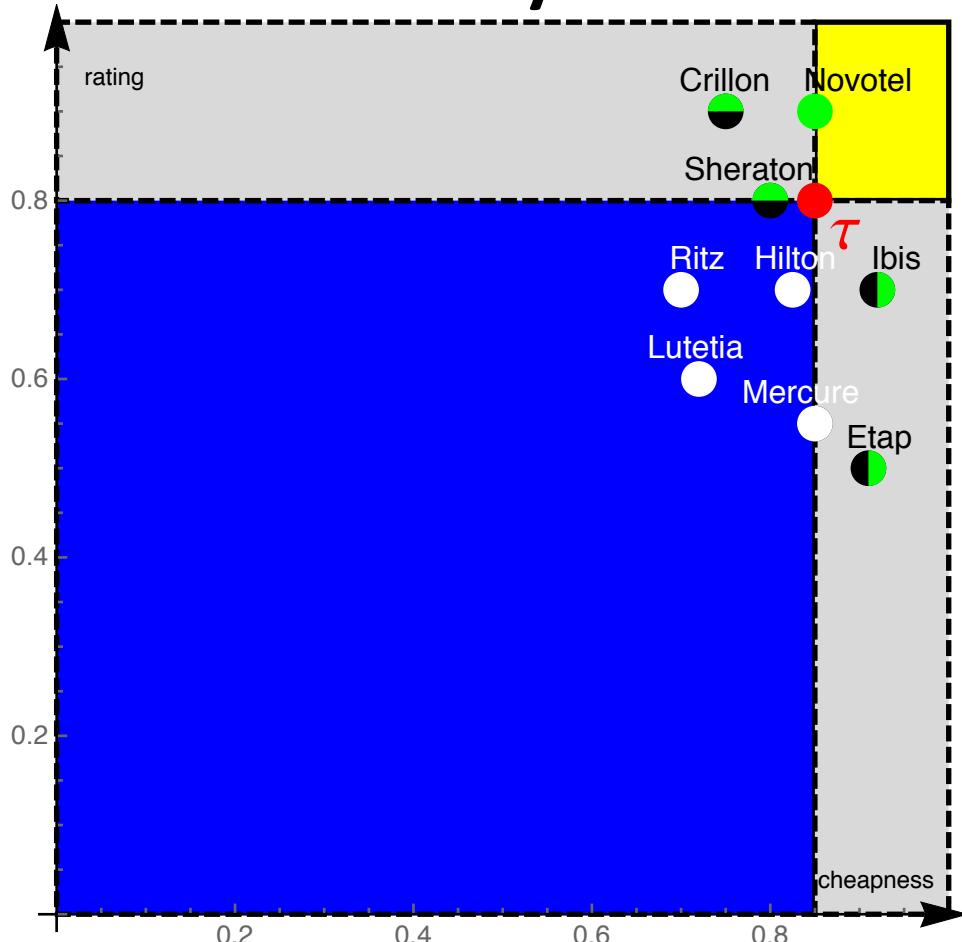
- The **threshold point (τ)** is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the **yellow region** (fully seen points) contains at least k points
- The **gray regions** contain the points seen in at least one ranking
- None of the points in the **blue region** (unseen points) can beat any point in the **yellow region**

Why does FA work?



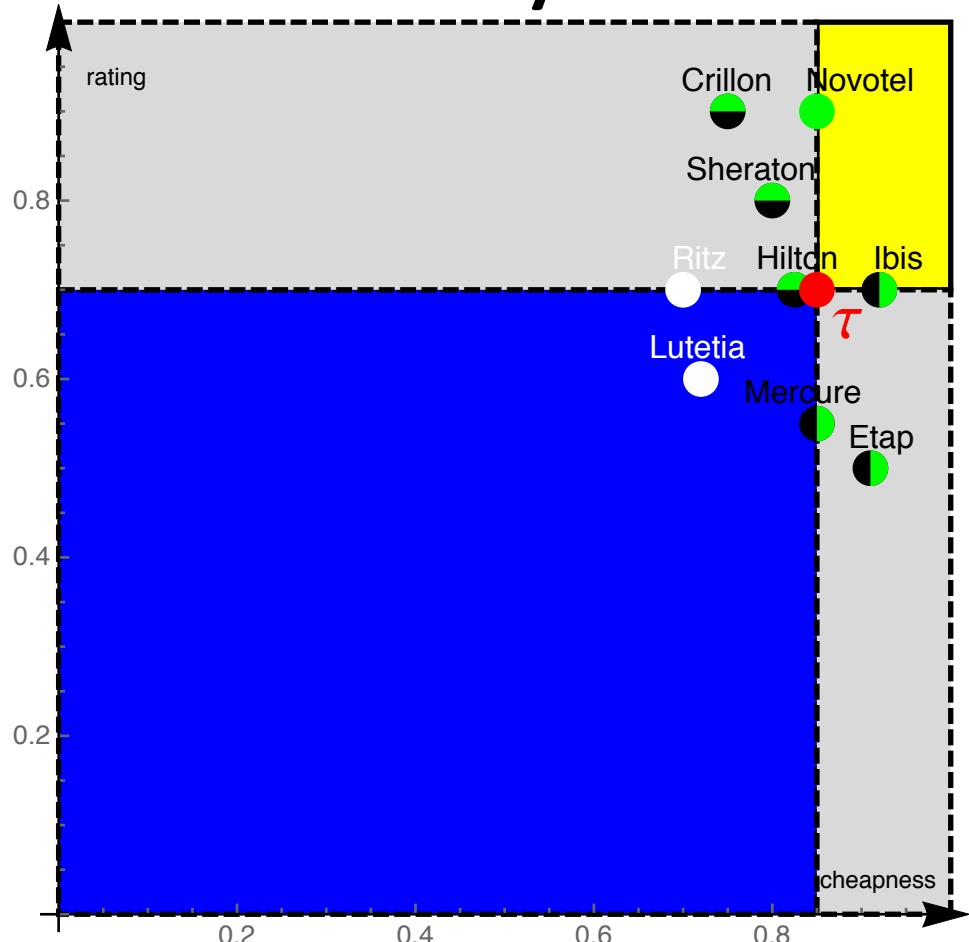
- The **threshold point (τ)** is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the **yellow region** (fully seen points) contains at least k points
- The **gray regions** contain the points seen in at least one ranking
- None of the points in the **blue region** (unseen points) can beat any point in the **yellow region**

Why does FA work?



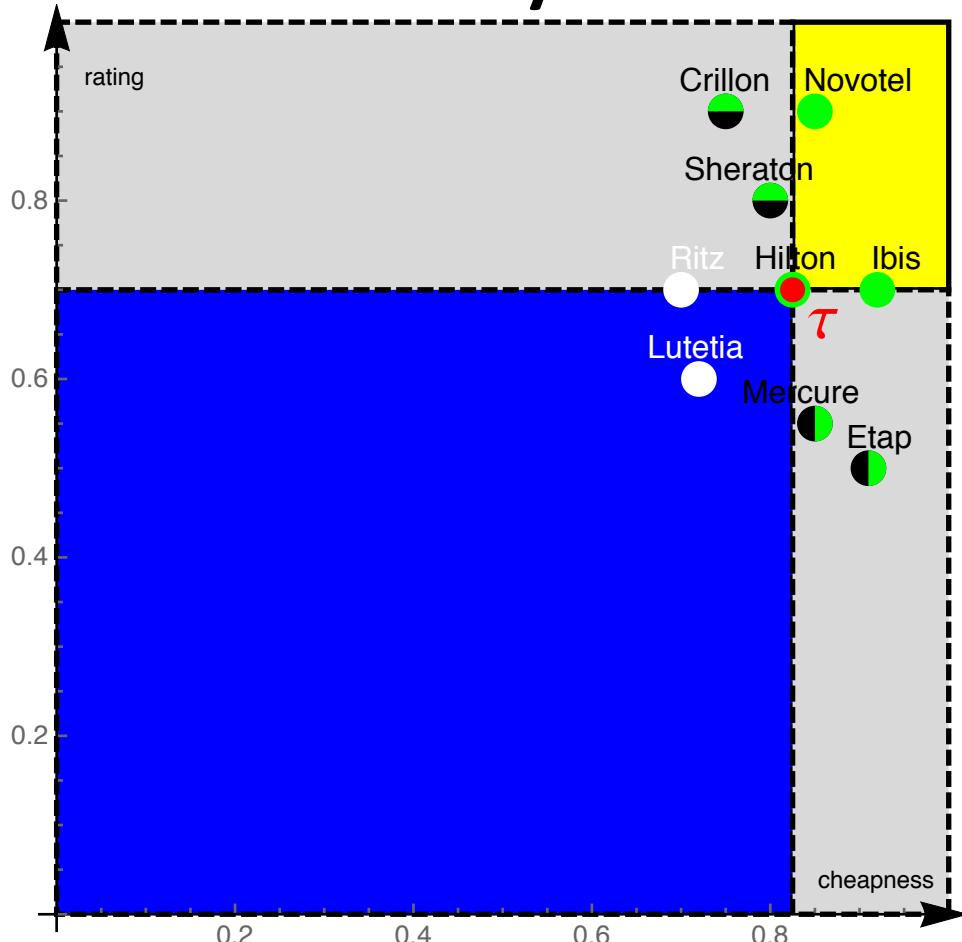
- The **threshold point (τ)** is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the **yellow region** (fully seen points) contains at least k points
- The **gray regions** contain the points seen in at least one ranking
- None of the points in the **blue region** (unseen points) can beat any point in the **yellow region**

Why does FA work?



- The **threshold point (τ)** is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the **yellow region** (fully seen points) contains at least **k** points
- The **gray regions** contain the points seen in at least one ranking
- None of the points in the **blue region** (unseen points) can beat any point in the **yellow region**

Why does FA work?



- The **threshold point (τ)** is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the **yellow region** (fully seen points) contains at least **k** points
- The **gray regions** contain the points seen in at least one ranking
- None of the points in the **blue region** (unseen points) can beat any point in the **yellow region**

Limits of FA

- Drawback: specific scoring function not exploited at all
 - In particular, the sorted+random access cost of FA is independent of the scoring function!
- Memory requirements can become prohibitive
 - FA has to buffer all the objects accessed through sorted access
- Small improvements are possible
 - E.g., by interleaving random accesses and score computation, which might save some random access
- Significant improvements require changing the stopping condition

Threshold Algorithm (TA)

[Fagin, Lotem, Naor, PODS 2001]

Input: integer k , a **monotone** function S combining ranked lists R_1, \dots, R_m

Output: the top k **<object, score>** pairs

1. Do a **sorted access** in parallel in each list R_i
2. For each object o , do **random accesses** in the other lists R_j , thus extracting score s_j
3. Compute overall score $S(s_1, \dots, s_m)$. If the value is among the k highest seen so far, remember o
4. Let s_{Li} be the last score seen under sorted access for R_i
5. Define threshold $T = S(s_{L1}, \dots, s_{Lm})$
6. If the score of the k -th object is worse than T , go to step 1
7. Return the current top- k objects

- TA is **instance-optimal** among all algorithms that use random and sorted accesses (FA is not)
 - The stopping criterion **depends** on the scoring function
- The authors of TA received the **Gödel prize in 2014** for the design of innovative algorithms

Example: hotels in Paris with TA

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
 - Alternate sorted access and random access
 - Maintain a threshold T
 - Stop when k objects are no worse than T

Hotels	Cheapness	Hotels	Rating	Top 2	Score
Ibis	.92	Crillon	.9		
Etap	.91	Novotel	.9		
Novotel	.85	Sheraton	.8		
Mercure	.85	Hilton	.7		
Hilton	.825	Ibis	.7		
Sheraton	.8	Ritz	.7		
Crillon	.75	Lutetia	.6		
...		...			

Threshold value: $T = ??$
 point: $\tau = (??, ??)$

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$

Hotels	Cheapness	Hotels	Rating
Ibis	.92	Crillon	.9
Etap	.91	Novotel	.9
Novotel	.85	Sheraton	.8
Mercure	.85	Hilton	.7
Hilton	.825	Ibis	.7
Sheraton	.8	Ritz	.7
Crillon	.75	Lutetia	.6
...		...	

Top 2	Score
Crillon	.825
Ibis	.81

Threshold
value: $T = .91$
point: $\tau = (.92, .9)$

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Then make a **random access** for each new hotel

Hotels	Cheapness	Hotels	Rating
Ibis	.92	Crillon	.9
Etap	.91	Novotel	.9
Novotel	.85	Sheraton	.8
Mercure	.85	Hilton	.7
Hilton	.825	Ibis	.7
Sheraton	.8	Ritz	.7
Crillon	.75	Lutetia	.6
...		...	

Top 2	Score
Novotel	.875
Crillon	.825

Threshold
value: $T = .905$
point: $\tau = (.91, .9)$

- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Make one sorted access at a time in each list
 - Then make a **random access** for each new hotel



The diagram illustrates a filtering process. On the left, a main table lists 10 hotels with their cheapness and rating. A secondary table on the right shows the top 2 entries from the main table, ordered by score. A green box on the right specifies a threshold value $T = .825$ and a point $\tau = (.85, .8)$.

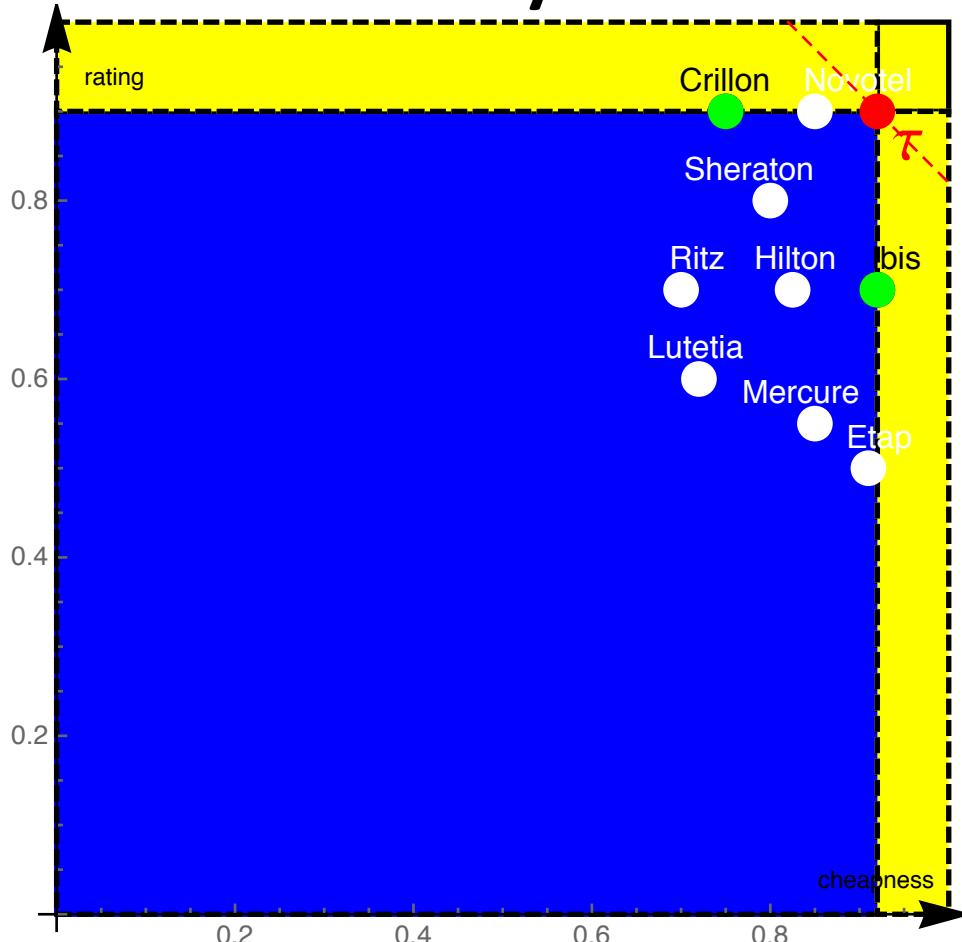
Hotels	Cheapness	Hotels	Rating
Ibis	.92	Crillon	.9
Etap	.91	Novotel	.9
Novotel	.85	Sheraton	.8
Mercure	.85	Hilton	.7
Hilton	.825	Ibis	.7
Sheraton	.8	Ritz	.7
Crillon	.75	Lutetia	.6
...		...	

Top 2	Score
Novotel	.875
Crillon	.825

Threshold
 value: $T = .825$
 point: $\tau = (.85, .8)$

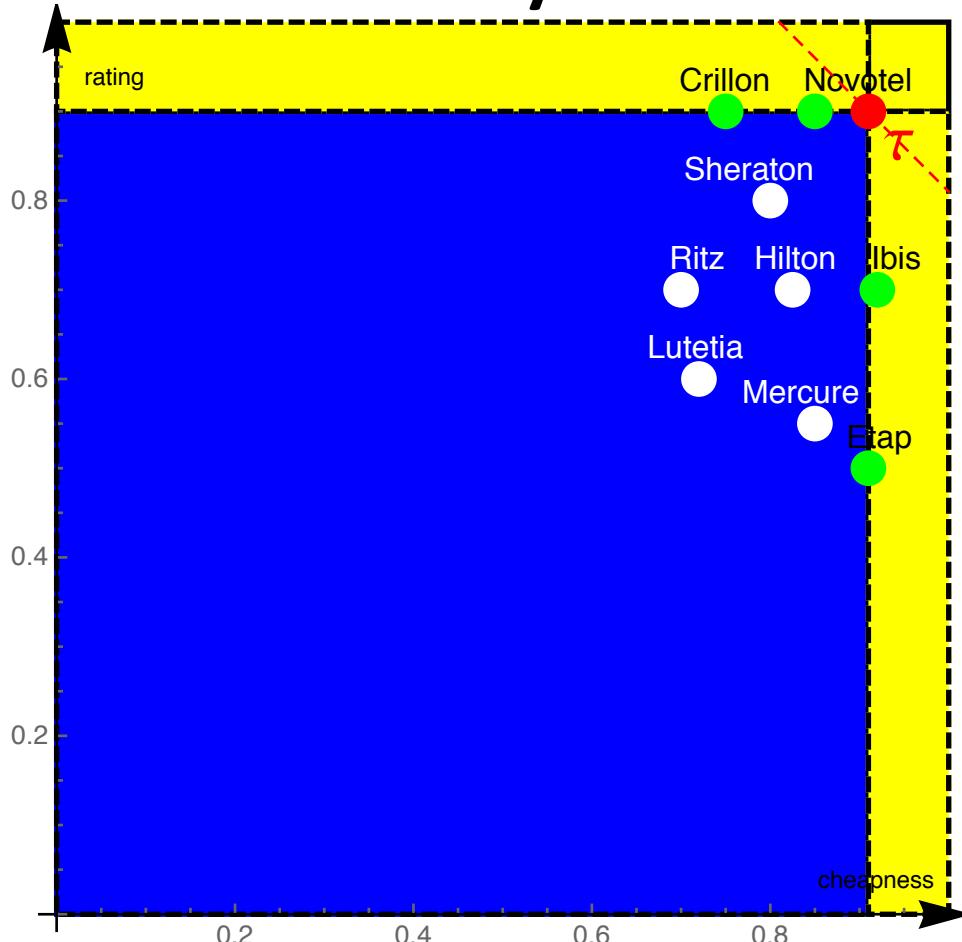
- Query: hotels with best price and rating
 - Scoring function: $0.5 * \text{cheapness} + 0.5 * \text{rating}$
- Strategy:
 - Stop when the score of the k -th hotel is no worse than the threshold

Why does TA work?



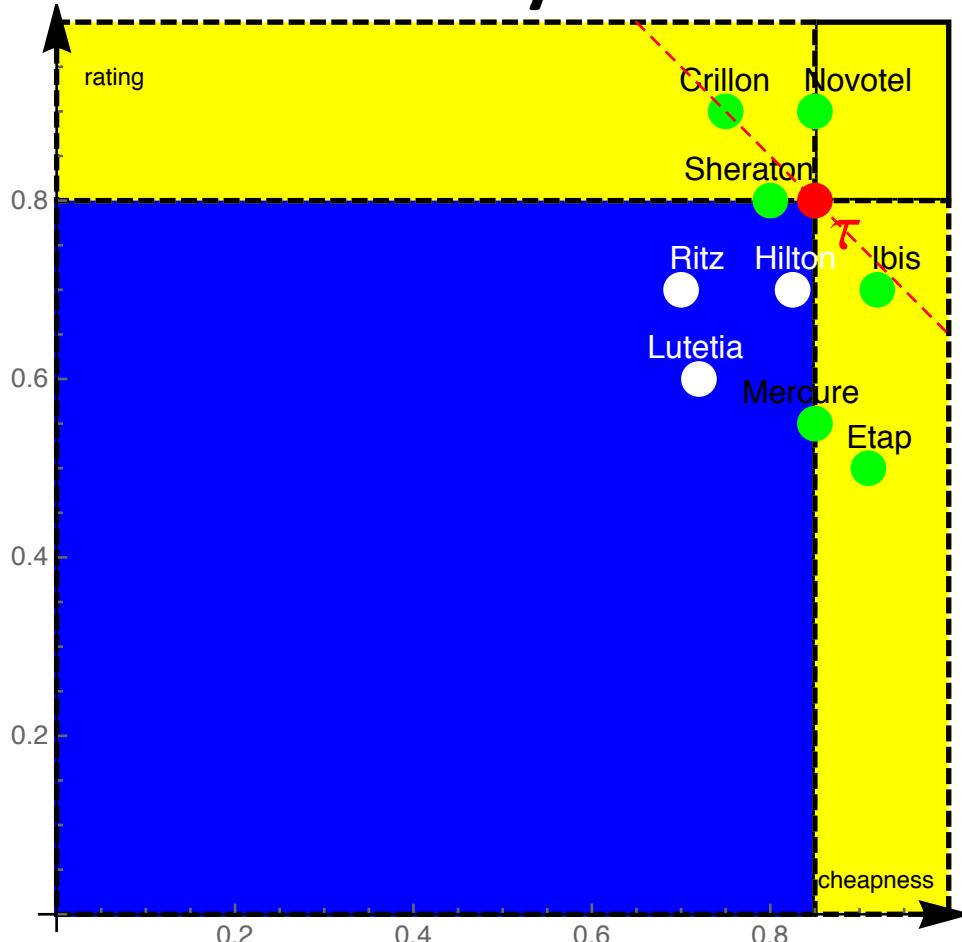
- τ is the **threshold point**
- FA stops when the **yellow region** (fully seen points) contains at least k points at least as good as τ
- None of the points in the **blue region** (unseen points) can beat τ
- The dashed **red line** separates the region of points dominating τ from the rest
 - No point dominates τ

Why does TA work?



- τ is the threshold point
- FA stops when the yellow region (fully seen points) contains at least k points at least as good as τ
- None of the points in the blue region (unseen points) can beat τ
- The dashed red line separates the region of points dominating τ from the rest
 - Still no point dominates τ

Why does TA work?



- τ is the threshold point
- FA stops when the yellow region (fully seen points) contains at least k points at least as good as τ
- None of the points in the blue region (unseen points) can beat τ
- The dashed red line separates the region of points dominating τ from the rest
 - Now, Novotel and Crillon dominate τ

Performance of TA: Cost model

- In general, TA performs much better than FA, since it can “adapt” to the specific scoring function
- In order to characterize the performance of TA, we consider the so-called **middleware cost**: $\text{cost} = \text{SA} * c_{\text{SA}} + \text{RA} * c_{\text{RA}}$, where:
 - SA (RA) is the total number of sorted (random) accesses
 - c_{SA} (c_{RA}) is the unitary (base) cost of a sorted (random) access
- In the basic setting, $c_{\text{SA}} = c_{\text{RA}}$ (=1, for simplicity)
- In other cases, base costs may differ
 - For web sources, usually $c_{\text{RA}} > c_{\text{SA}}$
 - limit case $c_{\text{RA}} = \infty$ (random access is impossible)
 - Some sources might not be accessible through sorted access
 - $c_{\text{SA}} = \infty$ (for instance, we do not have an index to process p_j)

The NRA algorithm: preliminaries

[Fagin, Lotem, Naor, PODS 2001]

- NRA (No Random Access) applies when random accesses cannot be executed
 - It returns the top-k objects, but their scores might be uncertain
 - This is to limit the cost of the algorithm
- Idea: maintain, for each object o retrieved by sorted access, a **lower bound** $S^-(o)$ and an **upper bound** $S^+(o)$ on its score
- NRA uses a buffer B with unlimited capacity, which is kept sorted according to decreasing lower bound values

OID	p1
o1	1.0
o7	0.9
...	...

OID	p2
o2	0.8
o3	0.75
...	...

OID	p3
o7	0.6
o2	0.6
...	...

$$S \equiv \text{SUM}$$

OID	Ibscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

The NRA algorithm

- Maintain a set Res of current best k objects
- Halt when no object o' not in Res can do better than any of the objects in Res
$$\forall o' \notin \text{Res}, \forall o \in \text{Res}: S^+(o') \leq S^-(o)$$
 - I.e., when $S^-(o)$ beats both
 - the max value of $S^+(o')$ among the objects in $B\text{-Res}$
 - and the score $S(\tau)$ of the threshold point τ (upper bound to unseen objects)

Input: integer $k \geq 1$, a **monotone** function S combining ranked lists R_1, \dots, R_m

Output: the top- k objects according to S

1. Make a sorted access to each list
2. Store in B each retrieved object o and maintain $S^-(o)$ and $S^+(o)$ and a threshold τ
3. Repeat from step 1 as long as $S^-(B[k]) < \max\{ \max\{S^+(B[i]), i > k\}, S(\tau) \}$

NRA: example

R₁

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
...	...

R₂

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
...	...

R₃

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
...	...

S ≡ SUM
k = 2

NRA: example

R_1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
...	...

R_2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
...	...

R_3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
...	...

$S \equiv \text{SUM}$

$k = 2$

$T = 2.4$

B (1st round)

OID	lbscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

$0.8 < \max\{2.4, 2.4\}$

NRA: example

R_1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
...	...

R_2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
...	...

R_3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
...	...

$S \equiv \text{SUM}$
 $k = 2$

$T = 2.25$

B (1st round)

OID	Ibscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

$0.8 < \max\{2.4, 2.4\}$

B (2nd round)

OID	Ibscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

$1.4 < \max\{2.35, 2.25\}$

NRA: example

R_1

OID	p1
o1	1.0
o7	0.9
o2	0.7
o6	0.2
...	...

R_2

OID	p2
o2	0.8
o3	0.75
o4	0.5
o1	0.4
...	...

R_3

OID	p3
o7	0.6
o2	0.6
o3	0.5
o5	0.1
...	...

$$S \equiv \text{SUM}$$

$$k = 2$$

$$T = 1.7$$

B (1st round)

OID	Ibscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

$$0.8 < \max\{2.4, 2.4\}$$

B (2nd round)

OID	Ibscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

$$1.4 < \max\{2.35, 2.25\}$$

B (3rd round)

OID	Ibscore	ubscore
o2	2.1	2.1
o7	1.5	2.0
o3	1.25	1.95
o1	1.0	2.0
o4	0.5	1.7

$$1.5 < \max\{2.0, 1.7\}$$

NRA: example

R_1

OID	p1	OID	p2	OID	p3
o1	1.0	o2	0.8	o7	0.6
o7	0.9	o3	0.75	o2	0.6
o2	0.7	o4	0.5	o3	0.5
o6	0.2	o1	0.4	o5	0.1
...

R_2

OID	p1	OID	p2	OID	p3
o1	1.0	o2	0.8	o7	0.6
o7	0.9	o3	0.75	o2	0.6
o2	0.7	o4	0.5	o3	0.5
o6	0.2	o1	0.4	o5	0.1
...

R_3

OID	p1	OID	p2	OID	p3
o1	1.0	o2	0.8	o7	0.6
o7	0.9	o3	0.75	o2	0.6
o2	0.7	o4	0.5	o3	0.5
o6	0.2	o1	0.4	o5	0.1
...

$$S \equiv \text{SUM}$$

$$k = 2$$

B (1st round)

OID	Ibscore	ubscore
o1	1.0	2.4
o2	0.8	2.4
o7	0.6	2.4

$$T = 0.7$$

$$0.8 < \max\{2.4, 2.4\}$$

B (4th round)

OID	Ibscore	ubscore
o2	2.1	2.1
o7	1.5	1.9
o1	1.4	1.5
o3	1.25	1.45
o4	0.5	0.8
o6	0.2	0.7
o5	0.1	0.7

B (2nd round)

OID	Ibscore	ubscore
o7	1.5	2.25
o2	1.4	2.3
o1	1.0	2.35
o3	0.75	2.25

$$1.4 < \max\{2.35, 2.25\}$$

B (3rd round)

OID	Ibscore	ubscore
o2	2.1	2.1
o7	1.5	2.0
o3	1.25	1.95
o1	1.0	2.0
o4	0.5	1.7

$$1.5 < \max\{2.0, 1.7\}$$

$$1.5 \geq \max\{1.5, 0.7\}$$

NRA: observations

- NRA's cost does not grow monotonically with k , i.e.
 - it might be cheaper to look for the top- k objects rather than for the top- $(k-1)$ ones!

Example:

- $k = 1$: the winner is $o2$ ($S(o2) = 1.2$), since the score of $o1$ is $S(o1) = 1.0$ and that of all other objects is 0.6. NRA has to reach depth $N-1$ to halt
- $k = 2$: to discover that the top-2 objects are $o1$ and $o2$ only 3 rounds are needed

$S \equiv \text{SUM}$

R_1		R_2	
OID	p1	OID	p2
o1	1.0	...	0.3
o2	1.0
...	0.3	...	0.3
...	...	o2	0.2
...	0.3	o1	0

- NRA is **instance-optimal** among all algorithms that do not make random accesses
 - optimality ratio is m

The overall picture

Algorithm	scoring function	Data access	Notes
B_0	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of scoring function
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, no exact scores

Summary on top-k 1-1 join queries

- There are several algorithms to process a top-k 1-1 join query
- Common assumption: the scoring function S is **monotone**
- Simplest case: MAX
- FA's stopping condition just considers the local rankings of the objects rather than their partial scores
- TA's stopping condition is based on a threshold T , which provides an upper bound to the scores of all unseen objects
- TA is instance-optimal, FA isn't
- NRA does not execute random accesses at all

Ranking queries – main aspects

- Very **effective** in identifying the best objects
 - Wrt. a specific **scoring function**
- Excellent **control of the cardinality** of the result
 - k is an input parameter of a top- k query
- For a user, it is **difficult to specify** a scoring function
 - E.g., the weights of a weighted sum
- Computation is very **efficient**
 - E.g., $O(N \log k)$ for local, unordered datasets of N elements
 - Instance optimality in some settings
- Easy to express the **relative importance of attributes**

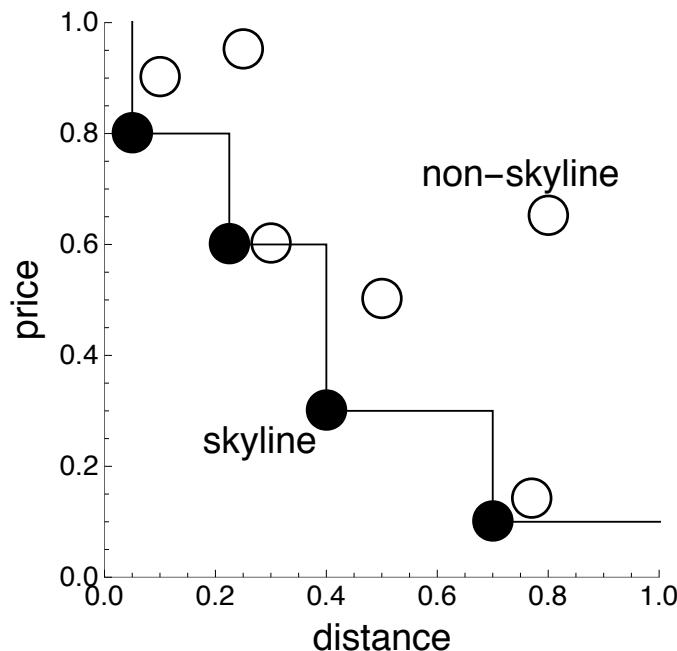
Skyline queries

Skylines: background

- Find good objects according to several different perspectives
 - e.g., attribute values A_1, \dots, A_m
- No need to specify weights!
 - based on the notion of **dominance**
- Tuple t **dominates** tuple s , indicated $t < s$, iff
 - $\forall i. 1 \leq i \leq m \rightarrow t[A_i] \leq s[A_i]$ (t is nowhere worse than s)
 - $\exists j. 1 \leq j \leq m \wedge t[A_j] < s[A_j]$ (and better at least once)
 - Typically, lower values are considered better
 - Opposite convention wrt. top-k queries, but it's just a convention that can be changed!

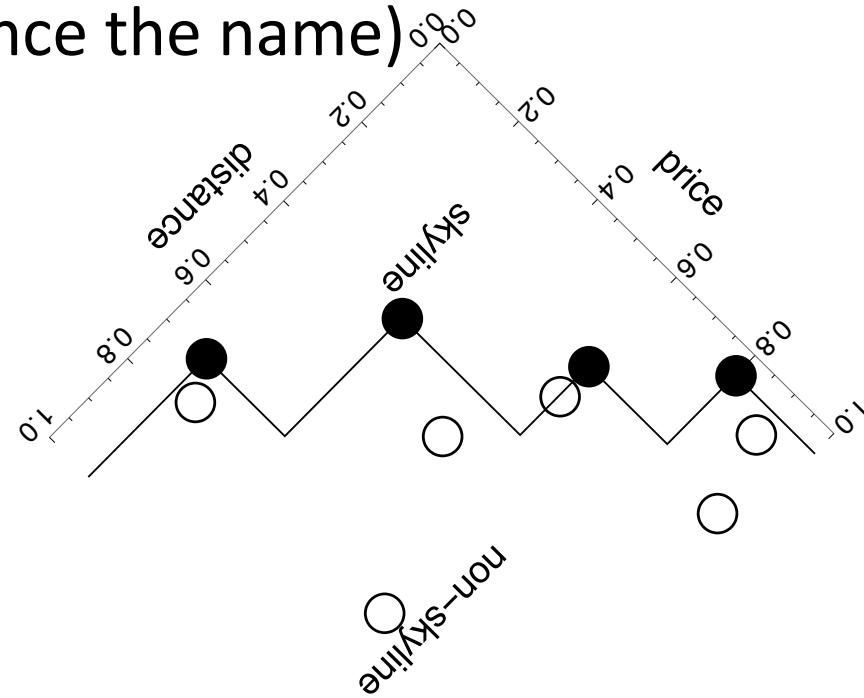
Skylines

- The **skyline** of a relation is the set of its non-dominated tuples. Aka:
 - Maximal vectors problem (computational geometry)
 - Pareto-optimal solutions (multi-objective optimization)



Skylines

- The **skyline** of a relation is the set of its non-dominated tuples. Aka:
 - Maximal vectors problem (computational geometry)
 - Pareto-optimal solutions (multi-objective optimization)
- In 2D, the shape resembles the contour of the dataset (hence the name)



Skylines: properties

- A tuple t is in the skyline iff it is the top-1 result w.r.t. at least one monotone scoring function
 - i.e., the skyline is the set of **potentially optimal tuples!**
- Skyline \neq Top- k query
 - there is no scoring function that, on all possible instances, yields in the first k positions the skyline points
 - based on the notion of **dominance**

Skyline queries in SQL

[Börzsönyi et al., ICDE 2001]

- Only a proposed syntax (not part of the standard):

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF],
          ..., dm [MIN | MAX | DIFF]
ORDER BY ...
```

- Example

```
SELECT * FROM Hotels WHERE city = 'Paris'
SKYLINE OF price MIN, distance MIN
```

Translation into naively nested SQL

```
SELECT * FROM Hotels WHERE city = 'Paris'  
SKYLINE OF price MIN, distance MIN
```

- Can be easily translated to actual SQL:

```
SELECT * FROM Hotels h  
WHERE h.city = 'Paris' AND NOT EXISTS (  
    SELECT * FROM Hotels h1  
    WHERE h1.city = h.city AND  
        h1.distance <= h.distance AND  
        h1.price     <= h.price AND  
        (h1.distance <  h.distance OR  
         h1.price     <  h.price))
```

- Very slow! We need a better way...

Skylines – Block Nested Loop (BNL)

[Börzsönyi et al., ICDE 2001]

Input: a dataset D of multi-dimensional points

Output: the skyline of D

1. Let $W = \emptyset$
2. for every point p in D
 3. if p not dominated by any point in W
 4. remove from W the points dominated by p
 5. add p to W
6. return W

- Computation is $O(n^2)$ where $n=|D|$
- Very inefficient for large datasets

Skylines – Sort-Filter-Skyline (SFS)

[Chomicki et al., ICDE 2003]

Input: a dataset D of multi-dimensional points

Output: the skyline of D

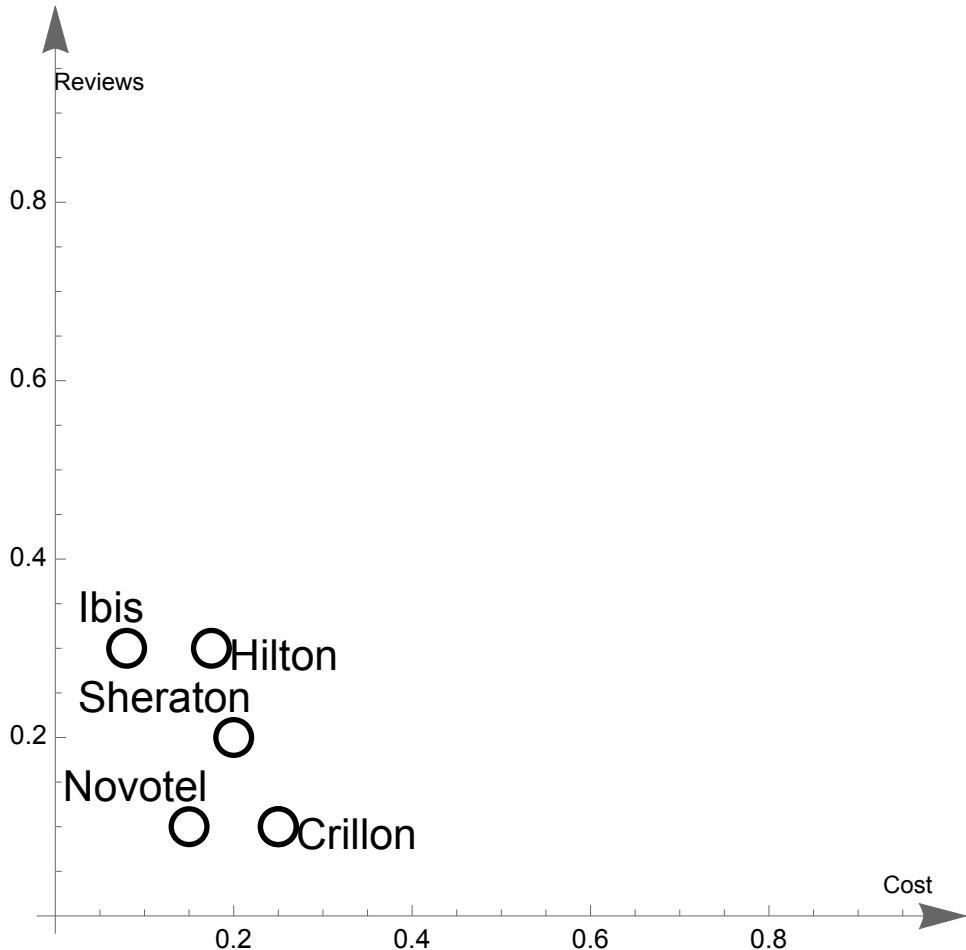
1. Let $S = D$ sorted by a monotone function of D 's attributes
2. Let $W = \emptyset$
3. for every point p in S
4. if p not dominated by any point in W
5. add p to W
6. return W

- Pre-sorting pays off for large datasets, thus SFS performs much better than BNL
 - If the input is sorted, then a later tuple cannot dominate any previous tuple!
 - Will never compare two non-skyline points
 - Can immediately output any points in W as part of the result
 - But still $O(n^2)$

Hotels	Cost	Reviews
Ibis	.08	.3
Novotel	.15	.1
Hilton	.175	.3
Crillon	.25	.1
Sheraton	.2	.2

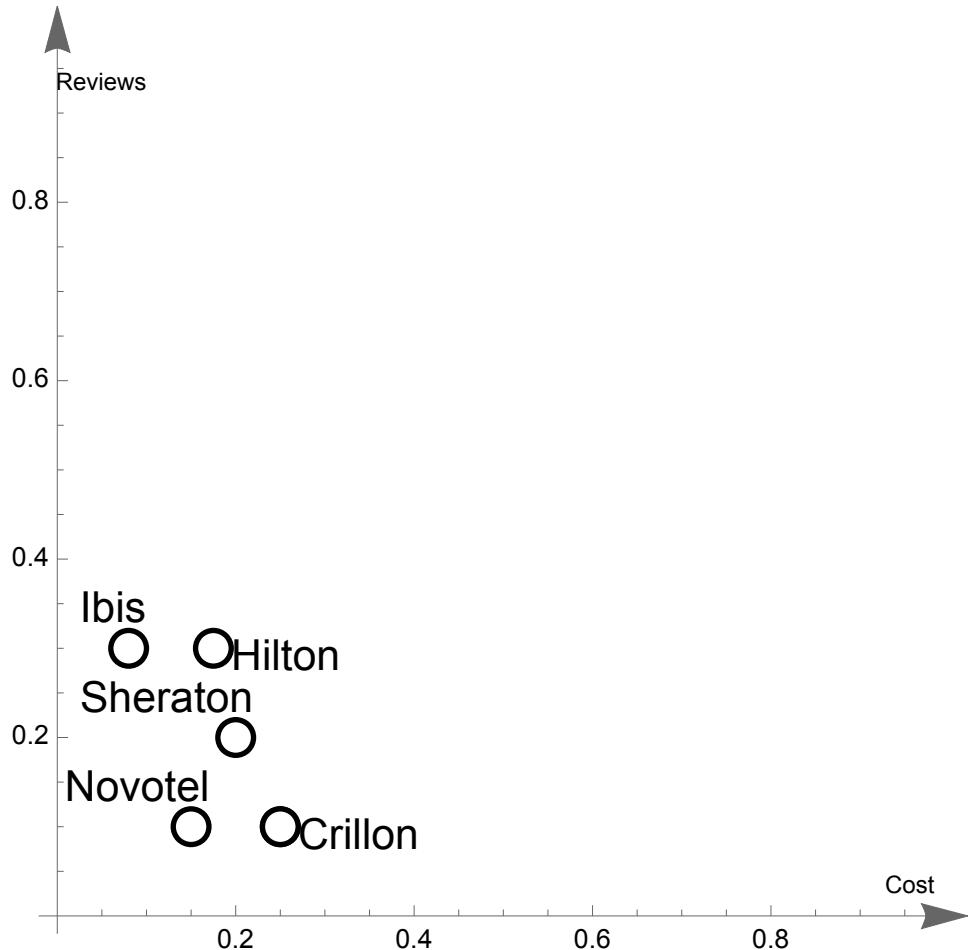
SFS

- Example dataset
 - (low values are good)



Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

- Sorted dataset
 - E.g., by Cost + Reviews

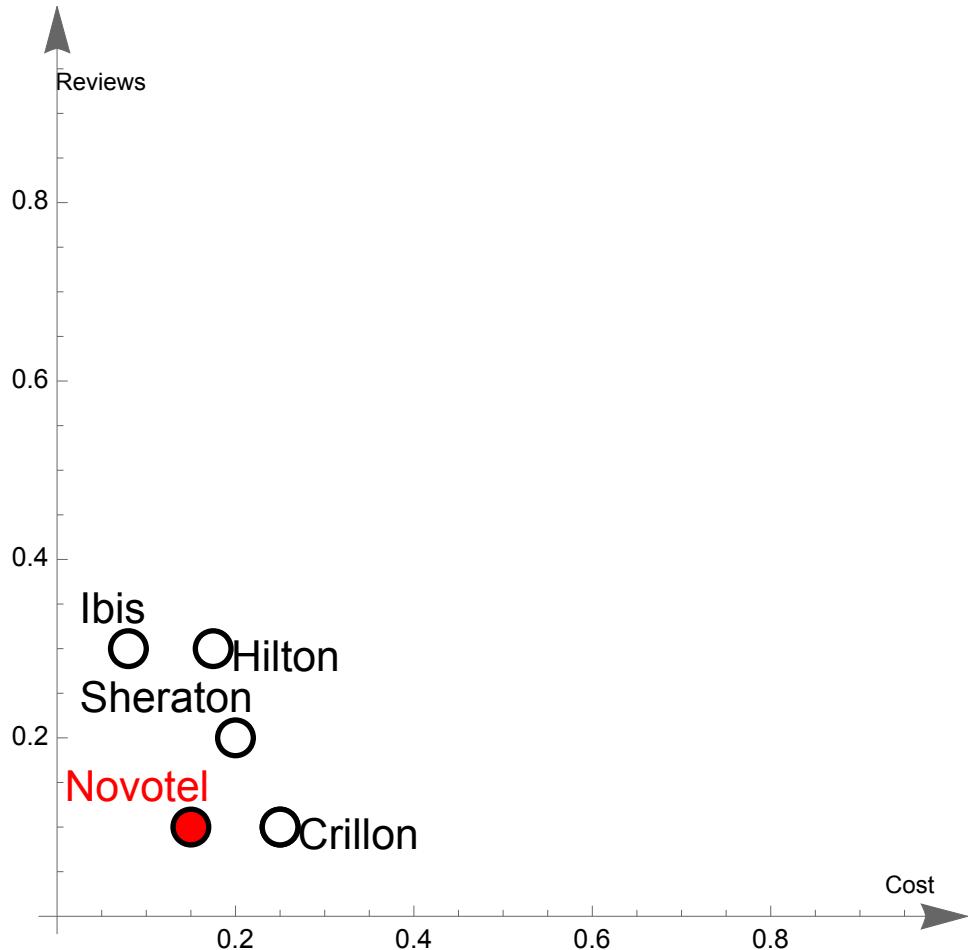


Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

Window

Hotels	Cost	Reviews
Novotel	.15	.1

- Sorted dataset
 - Add if not dominated by any point in the window



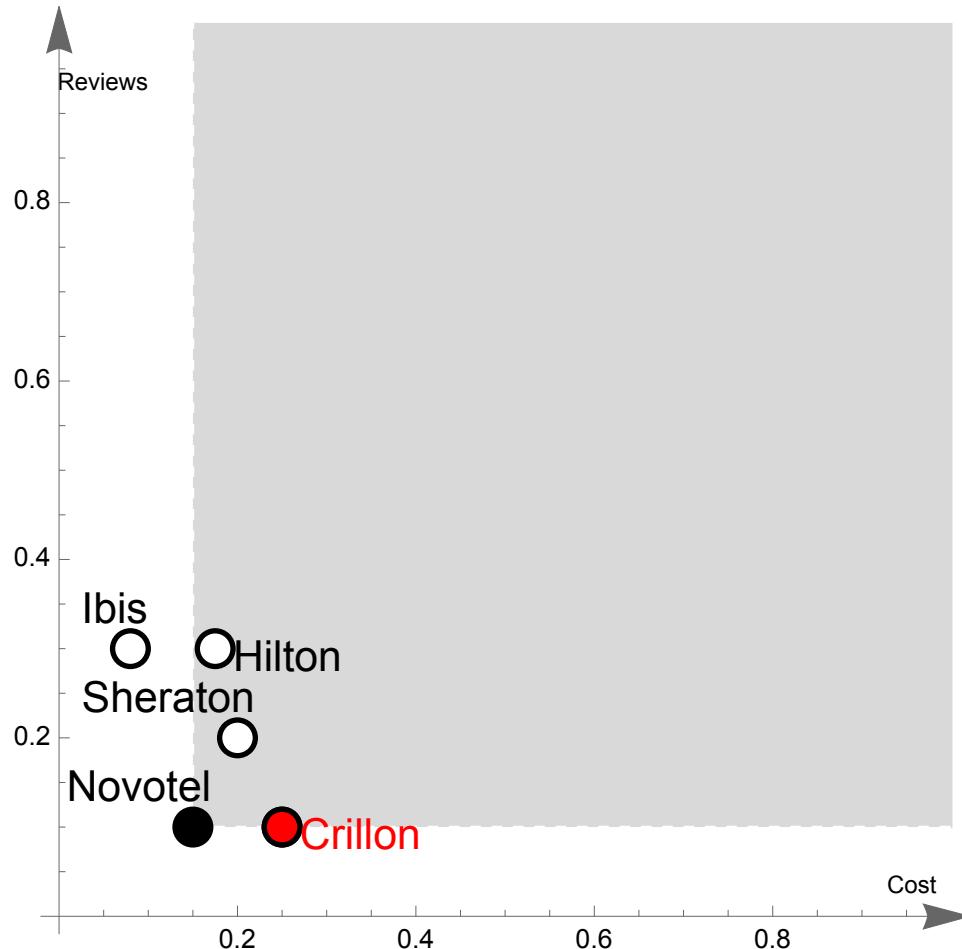
Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

Window

Hotels	Cost	Reviews
Novotel	.15	.1

Gray area: dominance region of the point(s) in the window

- Sorted dataset
 - Add if not dominated by any point in the window

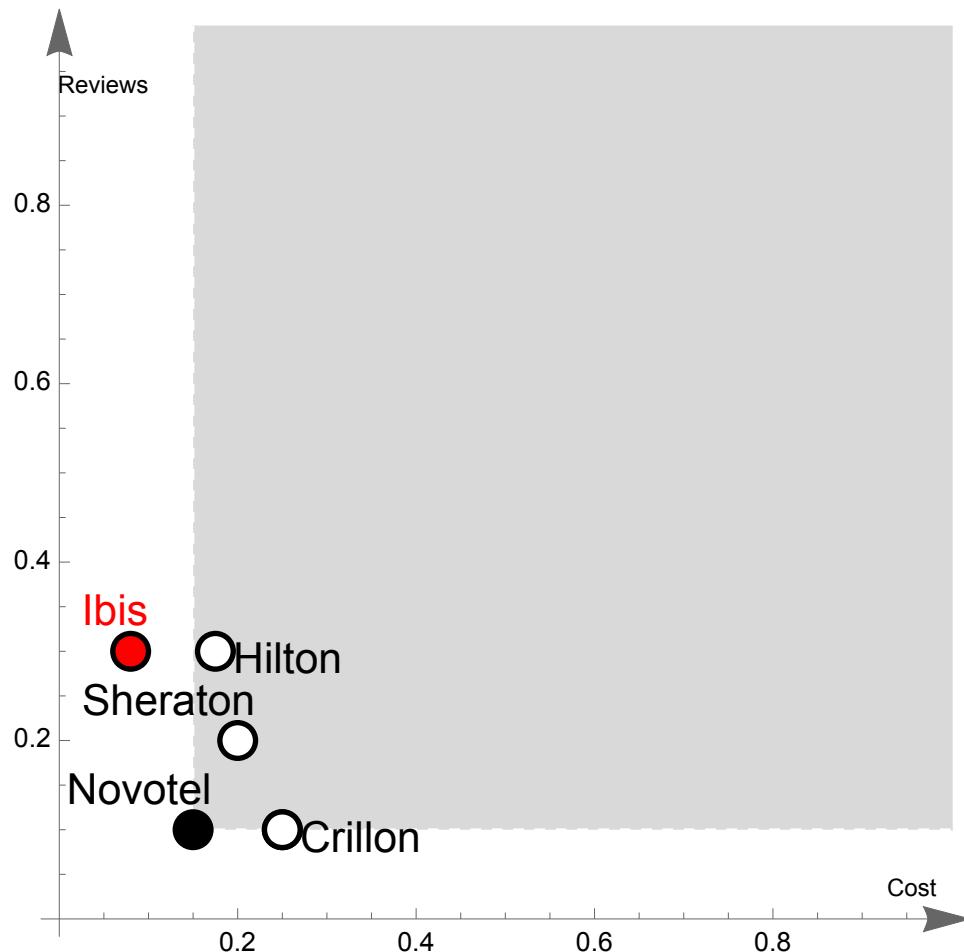


Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

Window

Hotels	Cost	Reviews
Novotel	.15	.1
Ibis	.08	.3

- Sorted dataset
 - Add if not dominated by any point in the window

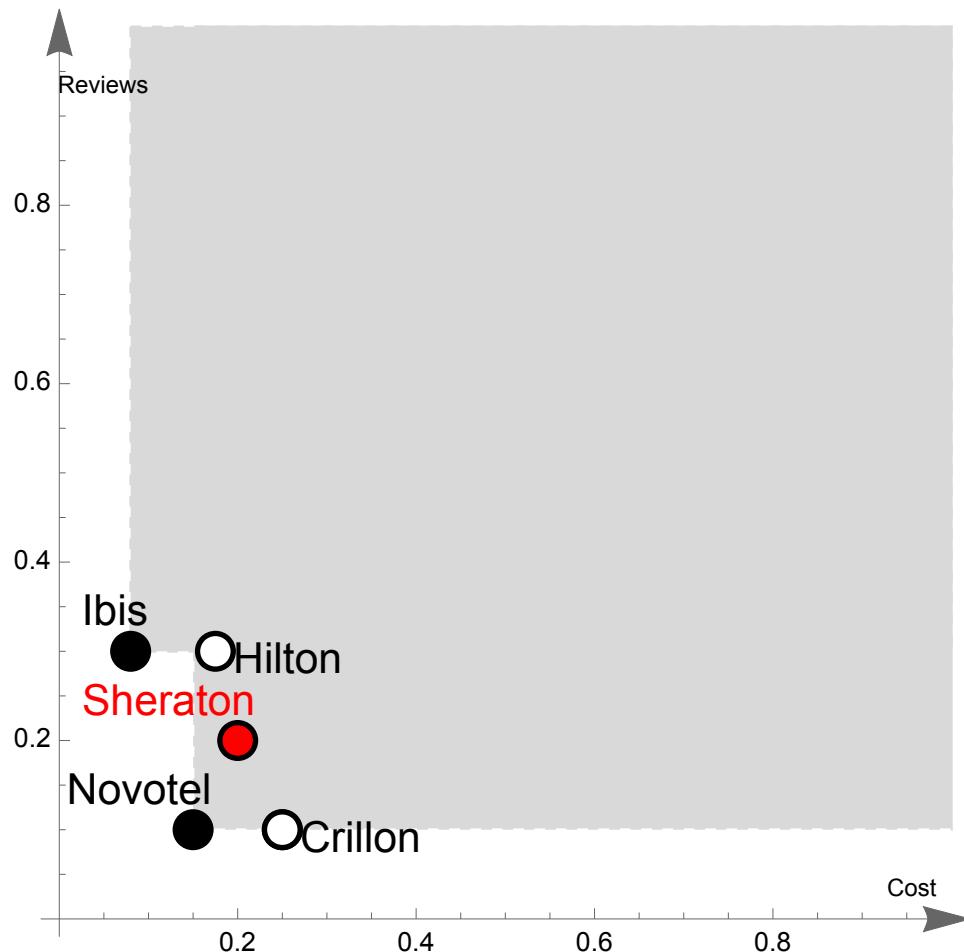


Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

Window

Hotels	Cost	Reviews
Novotel	.15	.1
Ibis	.08	.3

- Sorted dataset
 - Add if not dominated by any point in the window

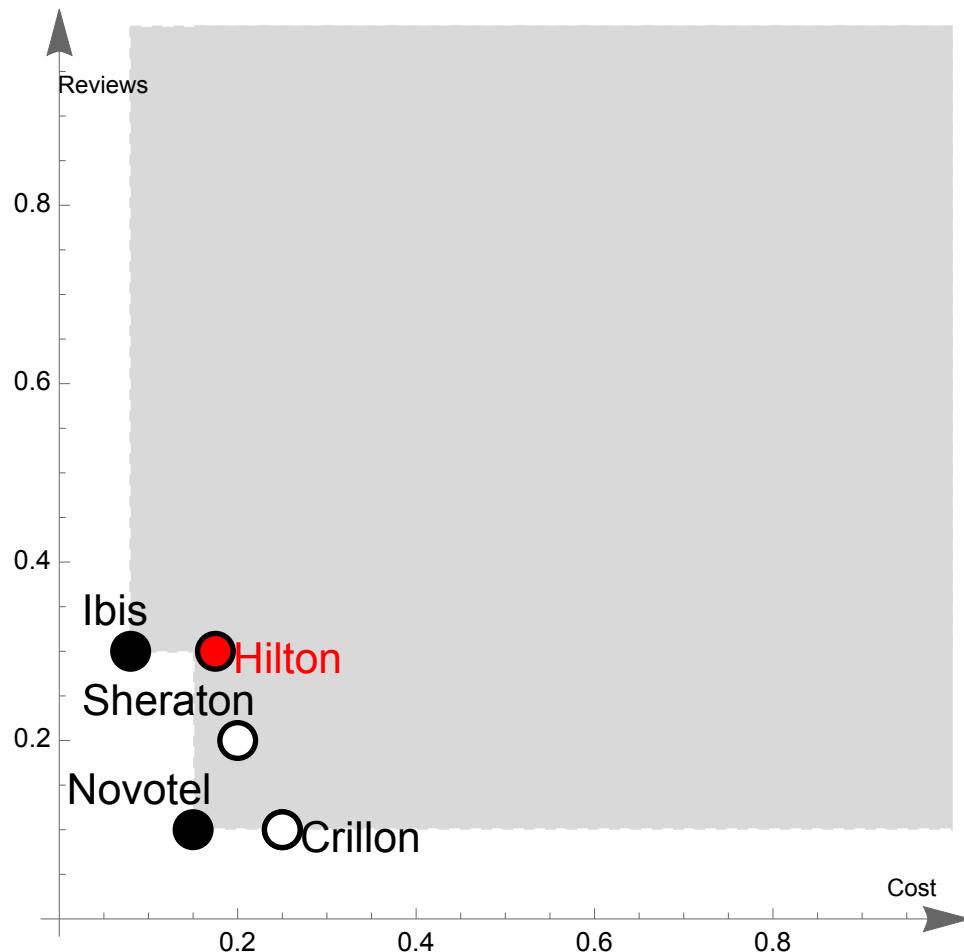


Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

Window

Hotels	Cost	Reviews
Novotel	.15	.1
Ibis	.08	.3

- Sorted dataset
 - Add if not dominated by any point in the window

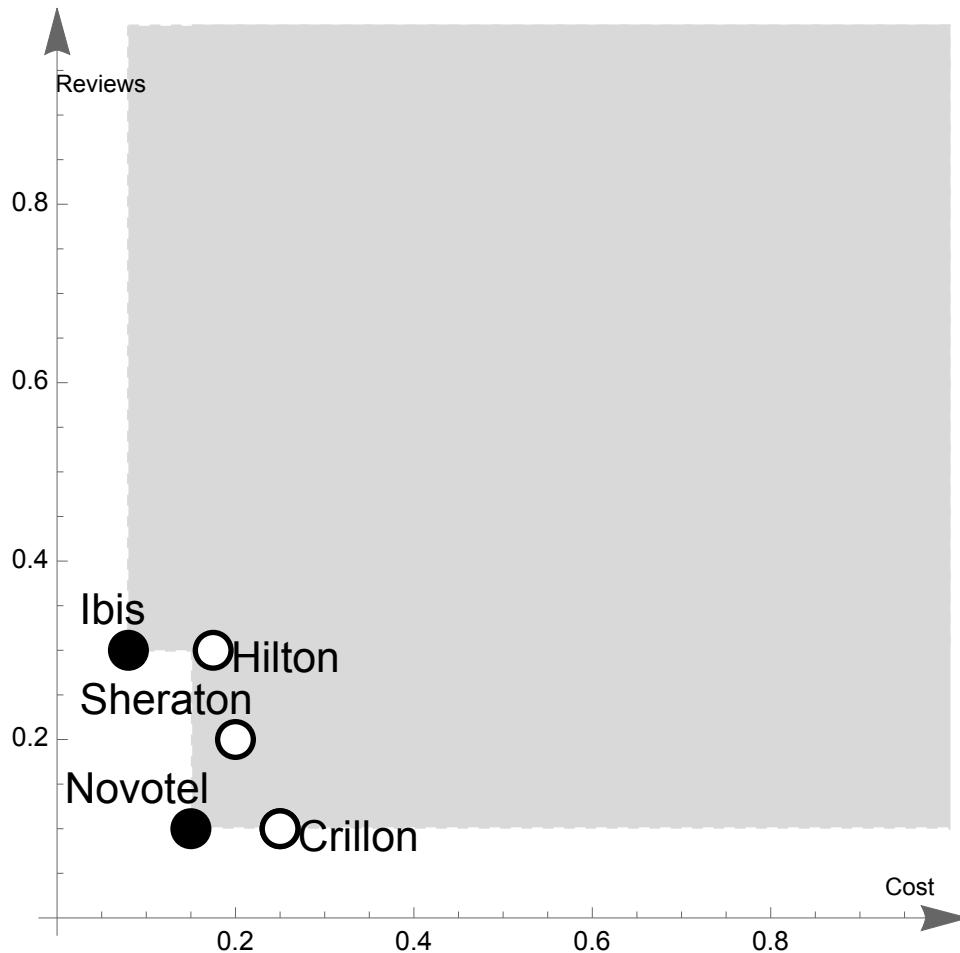


Hotels	Cost	Reviews
Novotel	.15	.1
Crillon	.25	.1
Ibis	.08	.3
Sheraton	.2	.2
Hilton	.175	.3

This is the skyline

Hotels	Cost	Reviews
Novotel	.15	.1
Ibis	.08	.3

- Sorted dataset
 - Add if not dominated by any point in the window

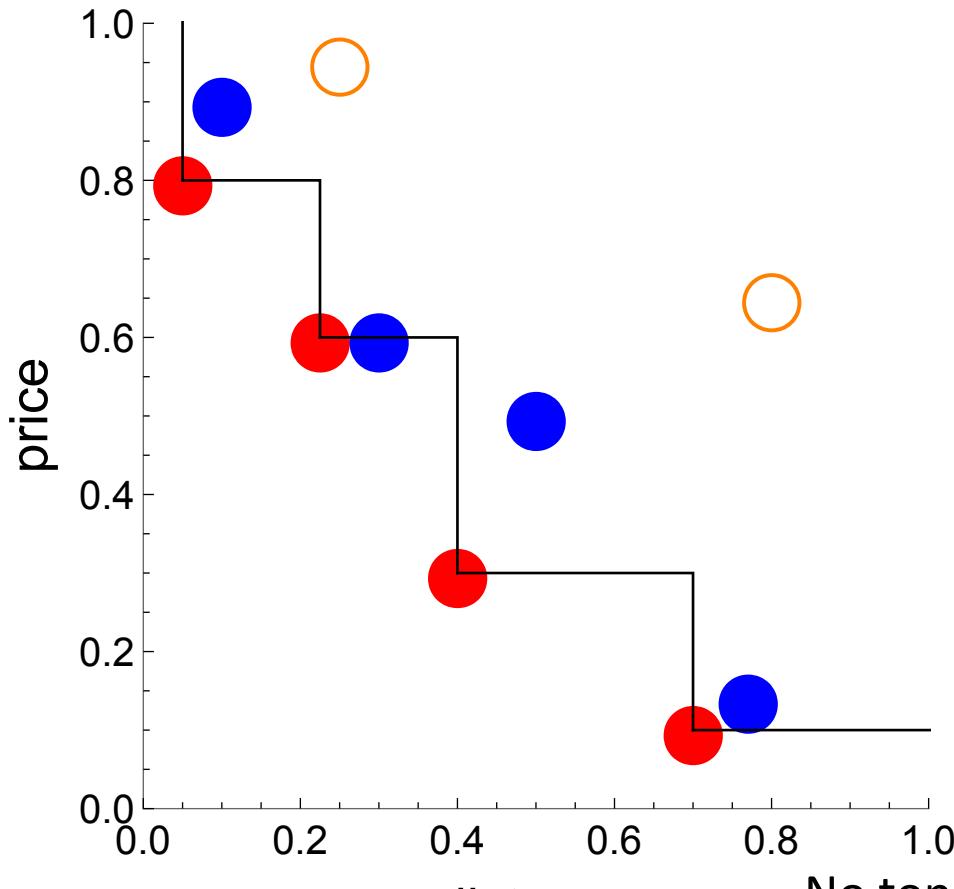


Skylines – main aspects

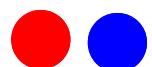
- Pros:
 - **Effective** in identifying potentially interesting objects if nothing is known about the preferences of a user
 - **Very simple** to use (no parameters needed!)
- Cons:
 - **Too many objects** returned for large, anti-correlated datasets
 - Computation is essentially quadratic in the size of the dataset (and thus **not so efficient**)
 - Agnostic wrt. known user preferences (e.g., price is more important than distance)
- Extension: **k-skyband** = set of tuples dominated by less than k tuples
 - Skyline = 1-skyband
 - Every top-k result set is contained in the k-skyband

Comparing different approaches

Example: skyline/k-skyband query



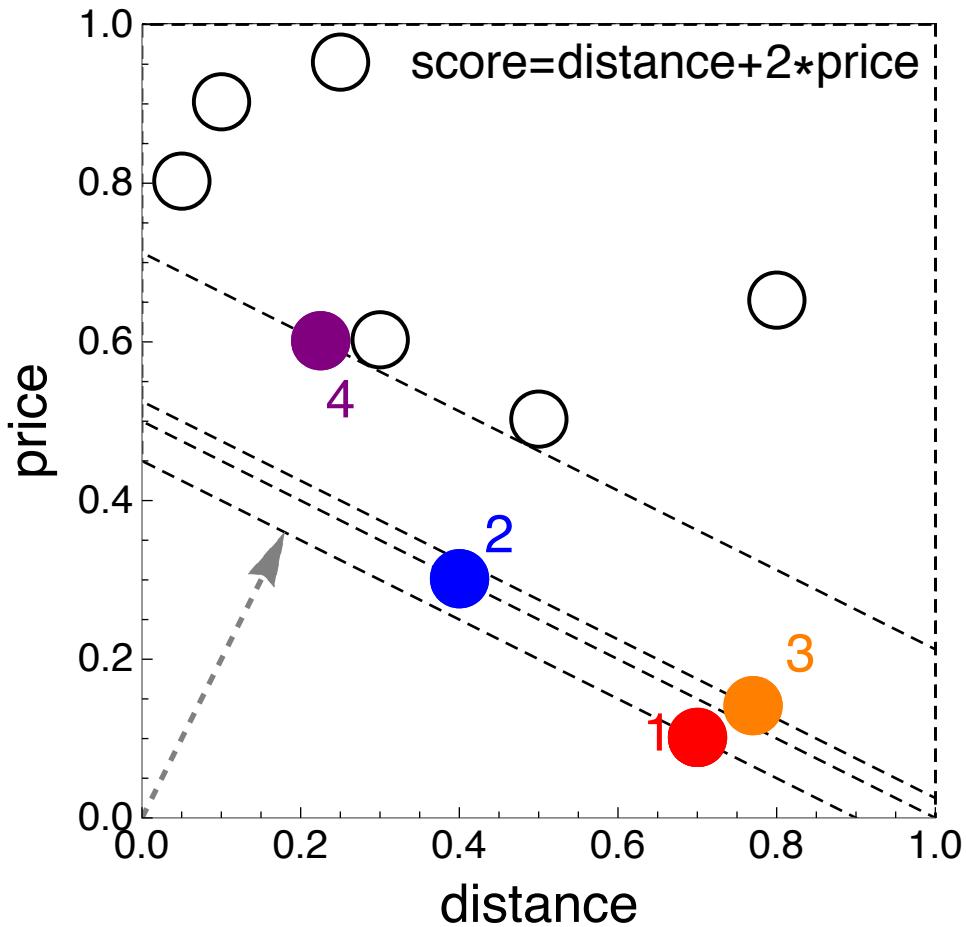
skyline



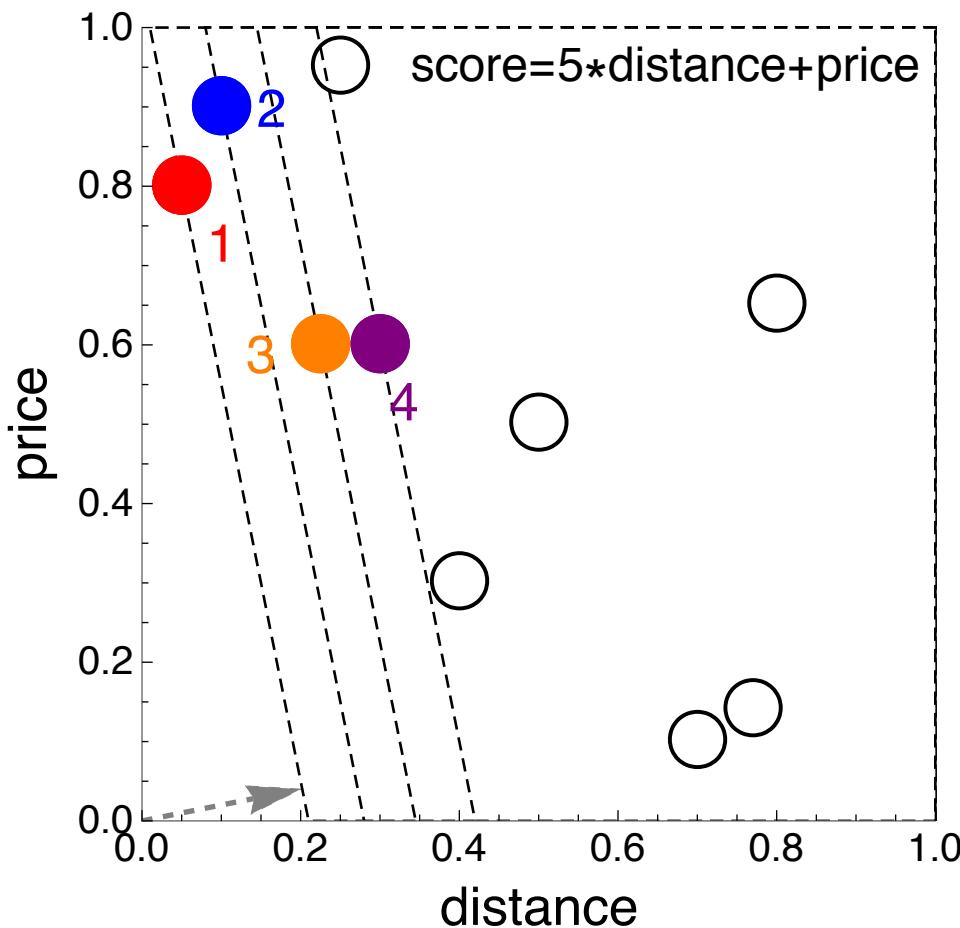
2-skyband = 3-skyband

No top-2 or top-3 query
will return a point

Example: ranking query



Example: another ranking query



Comparing different approaches

	Ranking queries	Skyline queries
Simplicity	No	Yes
Overall view of interesting results	No	Yes
Control of cardinality	Yes	No
Trade-off among attributes	Yes	No

Main References

Historical papers

- Jean-Charles de Borda
Mémoire sur les élections au scrutin. Histoire de l'Académie Royale des Sciences, Paris 1781
- Nicolas de Condorcet
Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix, 1785
- Kenneth J. Arrow
A Difficulty in the Concept of Social Welfare. Journal of Political Economy. 58 (4): 328–346, 1950

Rank aggregation and ranking queries

- Ronald Fagin, Ravi Kumar, D. Sivakumar
Efficient similarity search and classification via rank aggregation. SIGMOD Conference 2003: 301-312
- Ronald Fagin
Combining Fuzzy Information from Multiple Systems. PODS 1996: 216-226
- Ronald Fagin
Fuzzy Queries in Multimedia Database Systems. PODS 1998: 1-10
- Ronald Fagin, Amnon Lotem, Moni Naor
Optimal Aggregation Algorithms for Middleware. PODS 2001

Skylines and k-Skybands

- Stephan Börzsönyi, Donald Kossmann, Konrad Stocker
The Skyline Operator. ICDE 2001: 421-430
- Jan Chomicki, Parke Godfrey, Jarek Gryz, Dongming Liang
Skyline with Presorting. ICDE 2003: 717-719
- Dimitris Papadias, Yufei Tao, Greg Fu, Bernhard Seeger
Progressive skyline computation in database systems. ACM Trans. Database Syst. 30(1): 41-82 (2005)

Databases 2

Reliability Control

Topics

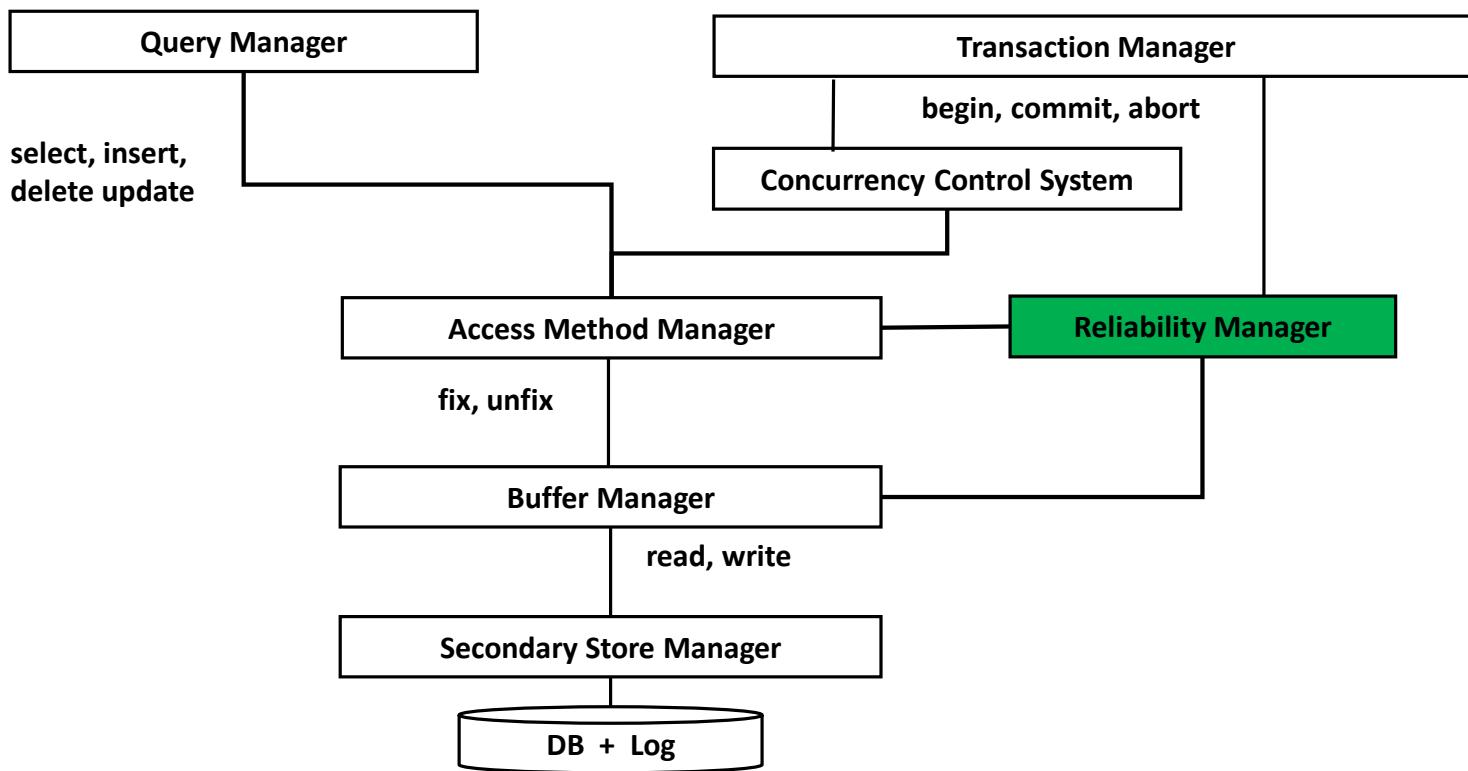
- Reliability definition and DBMS architecture
- Persistence of memory and backup
- Buffer management
- Reliable transaction management
- Log management
- Recovery after failures

Reliability

- *Reliability: The ability of an item to perform a required function under stated conditions for a stated period of time.*
[Reliability Definitions IEEE Dictionary of Electrical and Electronic Terms]
- In databases, reliability control ensures fundamental properties of transactions:
 - Atomicity: all or nothing semantics
 - Durability: persistence of effect
- DBMSs implement a specific architecture for reliability control
- The keys to database reliability are **stable memory** and **log management**

The Reliability Manager

- Realizes the transactional commands commit and abort
- Orchestrates read/write access to pages (both data and log)
- Handles recovery after failures

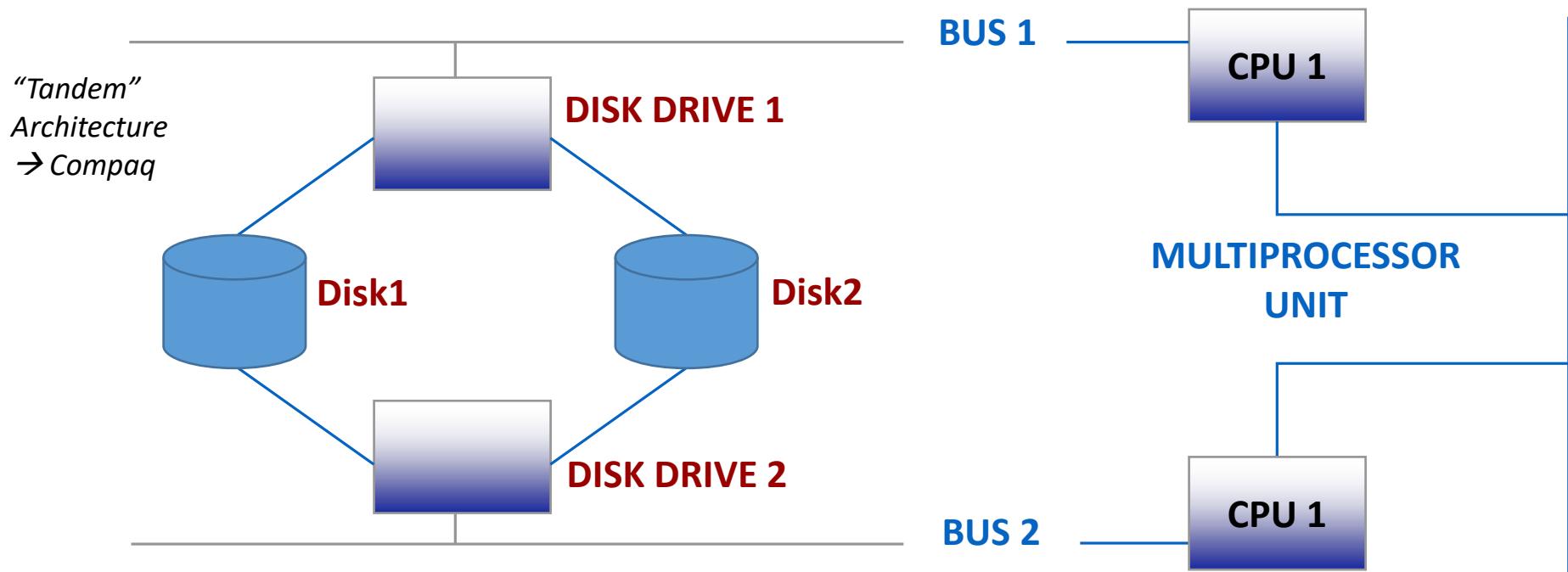


Persistence of Memory

- Durability implies a memory whose content lasts forever, which is an abstraction procedurally built on top of existing storage technology levels
- Main memory
 - Not persistent
- Mass memory
 - Persistent but can be damaged
- Stable memory
 - Cannot be damaged (clearly, this is an abstraction)
 - In practice stability = probability of failure close to 0
 - Keys to stability: replication and write protocols
- Failure of stable memory is the subject of the specific discipline of disaster recovery

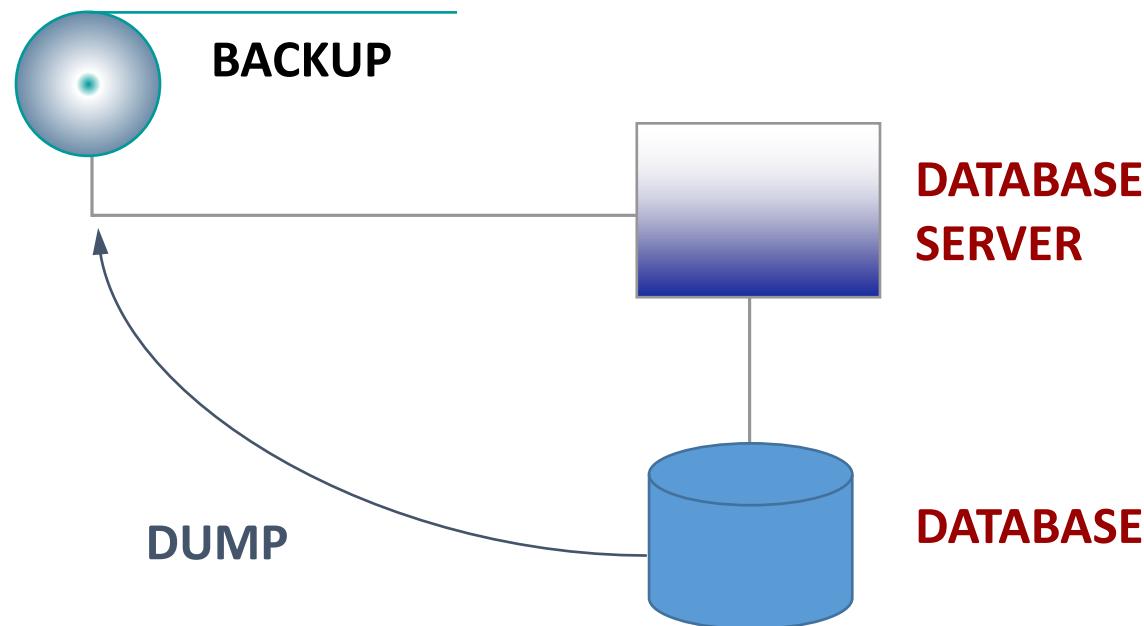
How to Guarantee Stable Memory

- On-line replication: mirroring of two disks
- The popular RAID (Redundant Array of Independent Disks) disk architecture is treated in the COMPUTING INFRASTRUCTURES course



How to Guarantee Stable Memory

- Off-line replication: "tape" units (backup units)



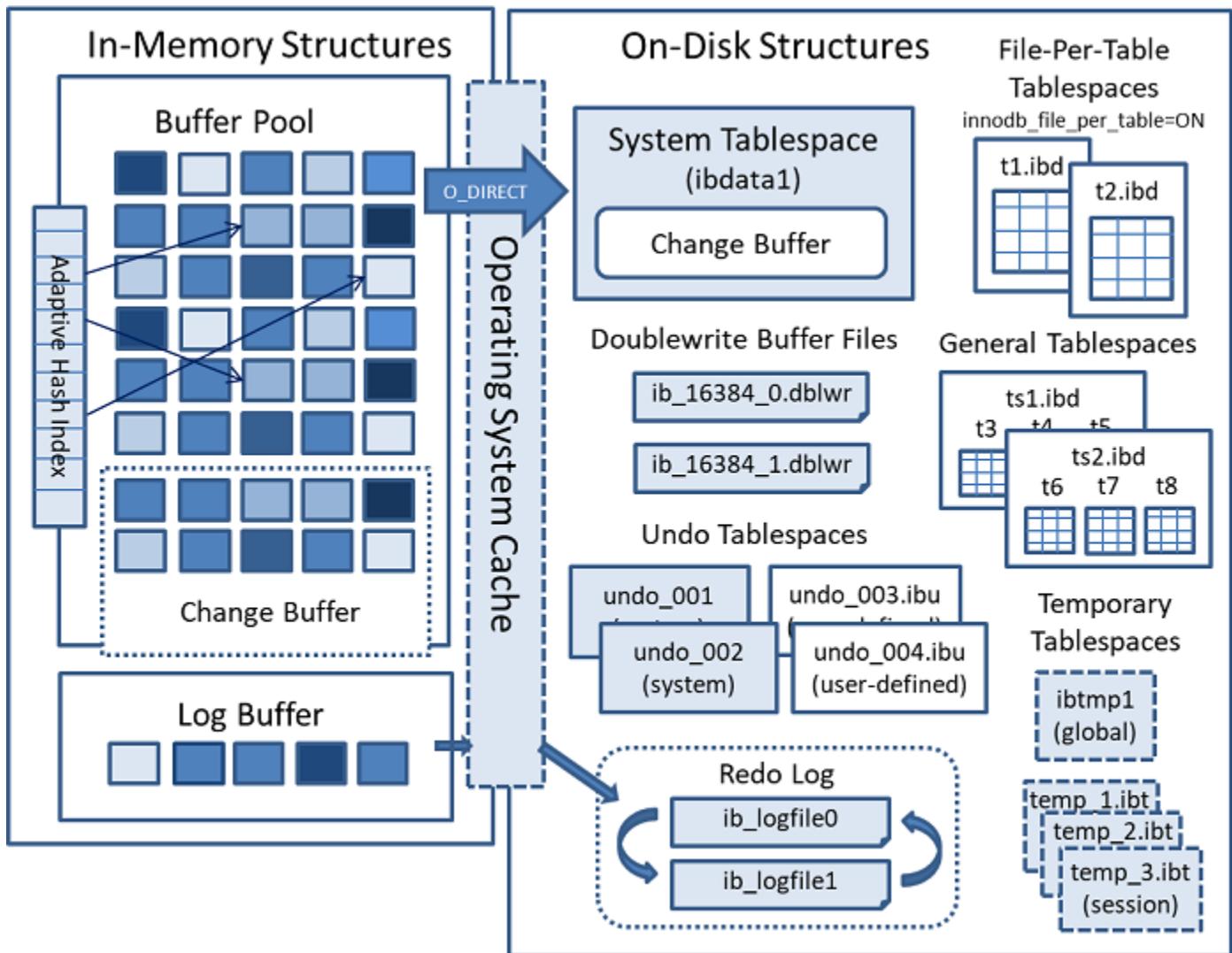
Stability or performance? Main Memory Management

- Rationale: reducing data access time without compromising memory stability
 - Use of buffer to cache data in faster memory
 - Deferred writing onto the secondary storage
- Organization
 - Buffer content accessed by page
 - Page may contain multiple rows and have
 - Transaction counter: how many transactions are using the page
 - Dirty flag: if true, page has been modified and must be aligned to secondary memory
- On dedicated DBMS servers, up to 80% of physical memory is often assigned to the buffer



Real DBMS architecture

- MySQL InnoDB data architecture



Use of Main Memory (buffer)

- Buffer management primitives:
 - fix
 - Responds to request by a transaction of loading of a page into the buffer. Returns a reference to the page and increments usage count
 - unfix
 - De-allocates page from the buffer and decrements usage count
 - force
 - Transfers synchronously a page from buffer to disk
 - setDirty
 - Modifies page status to denote that it has been changed
 - flush
 - Transfers asynchronously pages from buffer to disk, when page no longer needed (usage count == 0)



Execution of a fix primitive

- Searching for the target page
 - Search of the page in the buffer, if already there increment usage counter and return reference
 - Select a free (`usage_counter == 0`) page in the buffer (with FIFO or LRU policy); if found, return reference and increment usage counter. If `dirty_flag = true`, flush current page to disk before loading the new one
 - If no free page found, select page to de-allocate:
 - (**STEAL** policy) grab a victim page from an active transaction and flush it to disk; load new page increment usage counter and return reference
 - (**NO STEAL** policy) put the transaction in a wait list and manage the request when a page becomes free

Buffer Management Policies

- Write Policies: page writing to disk is normally asynchronous w.r.t. to transaction write operations
 - FORCE: pages are always transferred at commit
 - NO FORCE: transfer of pages can be delayed by the buffer manager
 - Normally buffer default configuration is:
 - NO STEAL, NO FORCE
- PRE-FETCHING (read-ahead in MySQL InnoDB)
 - anticipates loading of pages that are likely to be read
 - particularly effective in case of sequential reads
- PRE-FLUSHING
 - anticipates writing of de-allocated pages
 - effective for speeding up page fixing
- MySQL InnoDB LRU buffer management:
<https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html>

Buffer statistics in MySQL InnoDB

The screenshot shows the MySQL Workbench interface. The title bar says "MySQL Workbench" and "Local instance MySQL80". The menu bar includes File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The toolbar has various icons for database management. The Navigator pane shows a list of schemas: db2_schema, db_expense_management, db_gestione_spese, dbtest, jpa_book, sakila, sys, and world. The main area has tabs for "SQL File 8*", "SQL File 4*", "SQL File 5*", "mission", and "SQL File 8*". The "SQL File 8*" tab contains the query: "SELECT * FROM information_schema.INNODB_BUFFER_POOL_STATS". Below the query is a result grid with one row of data:

POOL_ID	POOL_SIZE	FREE_BUFFERS	DATABASE_PAGES	OLD_DATABASE_PAGES	MODIFIED_DATABASE_PAGES	PENDING_DECOMPRESS
0	512	251	256	0	0	0

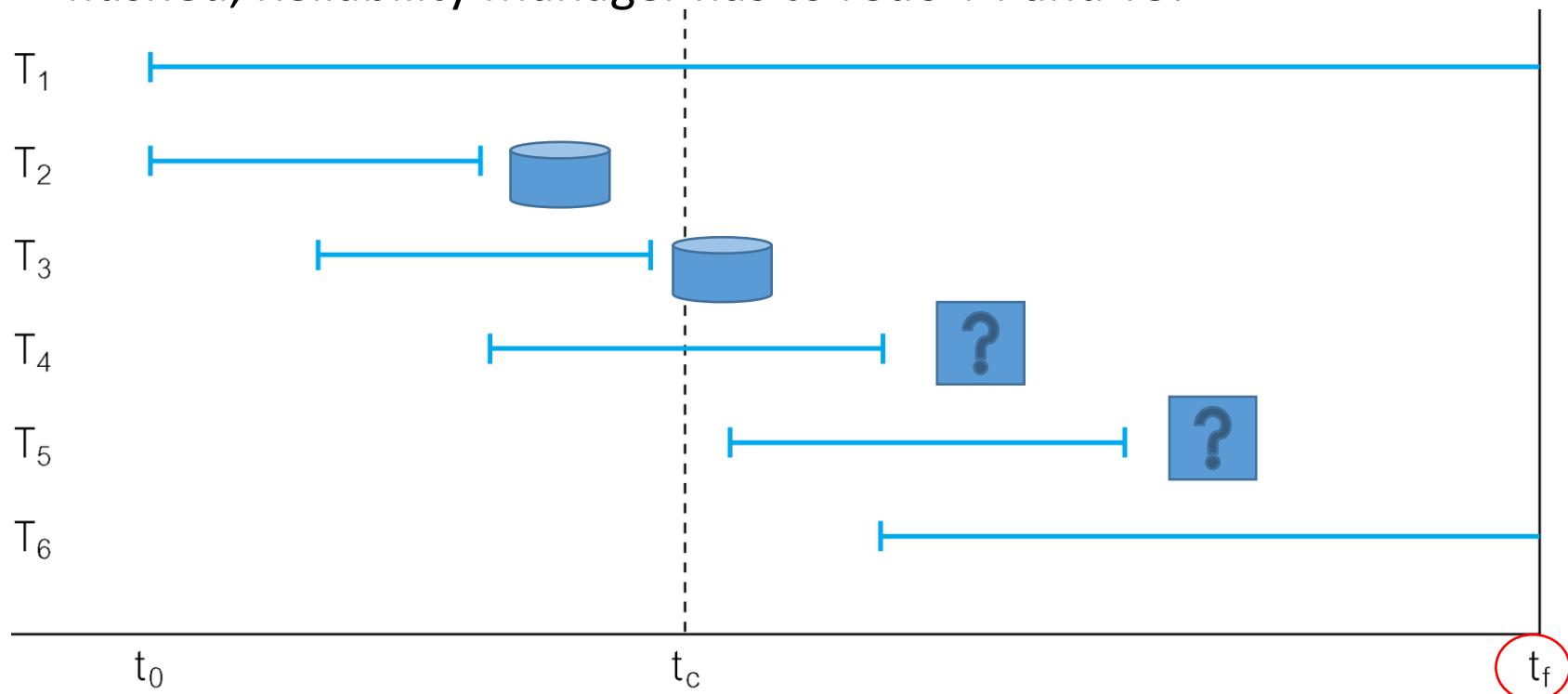
- To understand statistics and fine tune the buffer pool configuration:
 - <https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html>
 - <https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-buffer-pool-tables.html#innodb-information-schema-buffer-pool-stats-example>

Failure handling

- A transaction is an atomic transformation from an initial state into a final state
- Possible outcomes in absence/presence of failures:
 - **commit**: success
 - **rollback or fault before commit**: undo
 - **fault after commit**: redo
- Implications for recovery after failure
 - If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, Reliability Manager has to redo (rollforward) transaction updates.
 - If transaction had not committed at failure time, the Reliability Manager has to undo (rollback) any effects of that transaction for atomicity

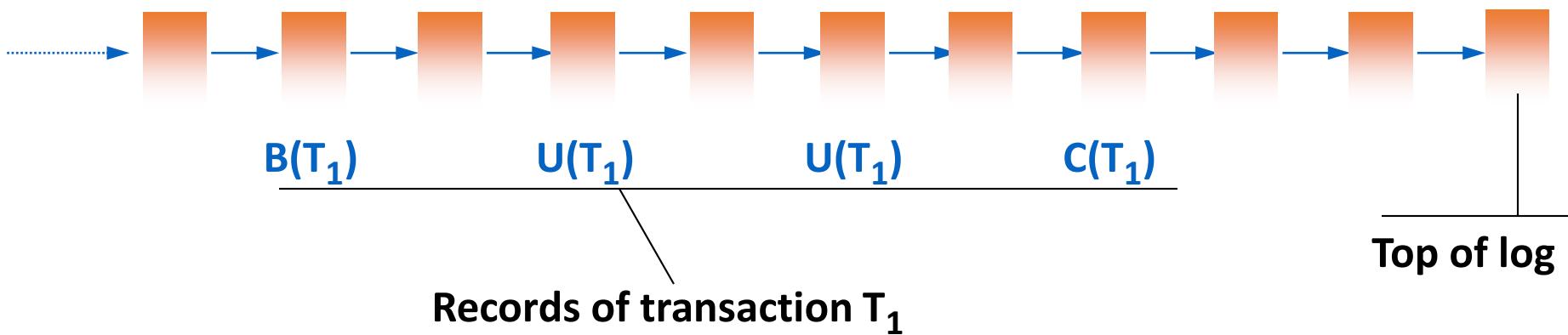
Transactions and recovery

- DBMS starts at time t_0 , but fails at time t_f .
- Assume data for transactions T_2 and T_3 have been written to secondary storage (committed and permanently stored).
- T_1 and T_6 have to be undone.
- In absence of information of whether modified pages have been flushed, Reliability Manager has to redo T_4 and T_5 .



Transaction Log

- A sequential file, made of records describing the actions carried out by the various transactions
- Written sequentially up to the top block (top = current instant)



Function of the Log

- It records on stable memory, in the form of state transitions, the actions carried out by the various transactions
 - if an UPDATE(U) operation transforms object O from value O₁ to value O₂
 - then the log records:
 - BEFORE-STATE(U) = O₁
 - AFTER-STATE(U) = O₂
- Logging INSERT and DELETE operations is identical to logging UPDATE operations, but
 - INSERT log record have no before state
 - DELETE log record have no after state

Example of MySQL log

binlogexample - Notepad

```
File Edit Format View Help
# original_commit_timestamp=1601194273790298 (2020-09-27 10:11:13.790298 W. Europe Summer Time)
# immediate_commit_timestamp=1601194273790298 (2020-09-27 10:11:13.790298 W. Europe Summer Time)
/*!80001 SET @@session.original_commit_timestamp=1601194273790298/*!*/;
/*!80014 SET @@session.original_server_version=80019/*!*/;
/*!80014 SET @@session.immediate_server_version=80019/*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 1156
#200927 10:11:13 server id 1 end_log_pos 1359 CRC32 0x9c06a82a      Query    thread_id=12      exec_time=0      error_code=0      Xid = 15
SET TIMESTAMP=1601194273/*!*/;
/*!80013 SET @@session.sql_require_primary_key=0/*!*/;
CREATE TABLE TableA (
    ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Val CHAR(1),
    IntVal INT
)
/*!*/;
# at 1359
#200927 10:11:13 server id 1 end_log_pos 1438 CRC32 0xcb7cbbbe      Anonymous_GTID  last_committed=5      sequence_number=6      rbr_
/*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED/*!*/;
# original_commit_timestamp=1601194273810786 (2020-09-27 10:11:13.810786 W. Europe Summer Time)
# immediate_commit_timestamp=1601194273810786 (2020-09-27 10:11:13.810786 W. Europe Summer Time)
/*!80001 SET @@session.original_commit_timestamp=1601194273810786/*!*/;
/*!80014 SET @@session.original_server_version=80019/*!*/;
/*!80014 SET @@session.immediate_server_version=80019/*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 1438
#200927 10:11:13 server id 1 end_log_pos 1519 CRC32 0x13b7c80e      Query    thread_id=12      exec_time=0      error_code=0
SET TIMESTAMP=1601194273/*!*/;
BEGIN
/*!*/;
# at 1519
#200927 10:11:13 server id 1 end_log_pos 1586 CRC32 0x01dc5fb2      Table map: `db2 schema`.`tablea` mapped to number 84
<                                     Ln 1, Col 1      100%      Windows (CRLF)      UTF-16 LE
```

This is what we did in the lesson on anomalies in MySQL!

Using the Log

- After: rollback-work or a failure that occurred before commit
 - **UNDO T1:** $O = O_1$
- After: a failure that occurred after commit
 - **REDO T1:** $O = O_2$
- **Idempotency** of UNDO and REDO:
 - $\text{UNDO}(T) = \text{UNDO}(\text{UNDO}(T))$
 - $\text{REDO}(T) = \text{REDO}(\text{REDO}(T))$
- Idempotency is necessary because the Reliability Manager may undo/redo operations twice, if its in doubt about the status of a write (flushed or not). Idempotency ensures that the effect is the same in any case

Types of Log Records

- Records concerning transactional commands:
 - **B(T)** , **C(T)** , **A(T)**
- Records concerning UPDATE, INSERT ans DELETE operations
 - **U(T,O,BS,AS)** , **I(T,O,AS)** , **D(T,O,BS)**
- Records concerning recovery actions
 - **DUMP** , **CKPT(T1,T2,...,Tn)**
- Record fields:
 - T_i: transaction identifier
 - O: object identifier
 - BS, AS: before state, after state

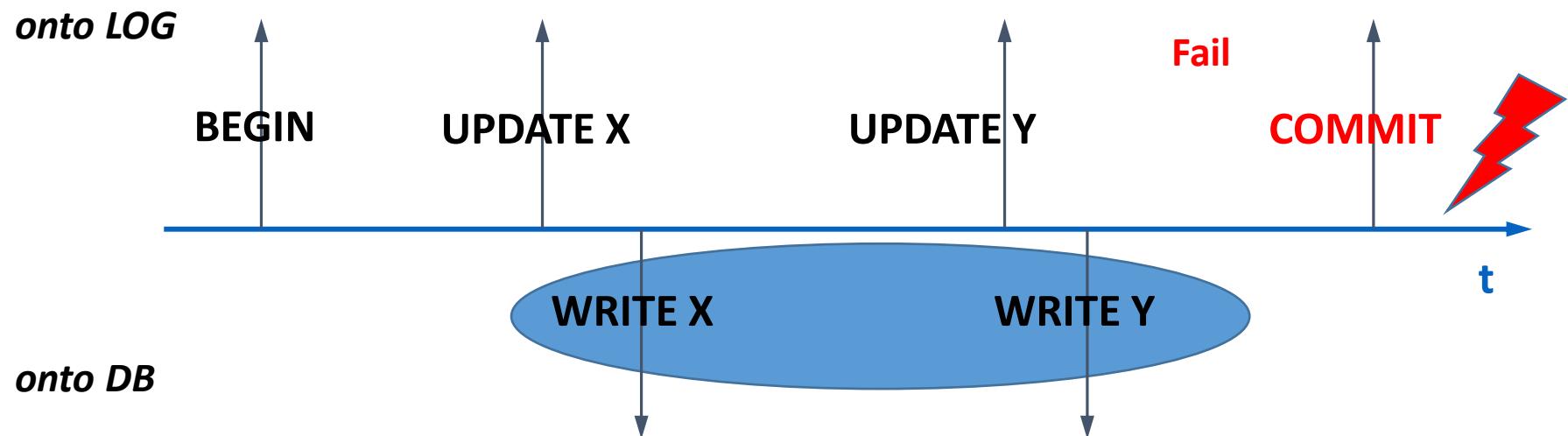
```
BINLOG '  
YU1wXxMBAAAAQwAAAB8aAAAAAFgAAAAAAAECmRiMl9zY2h1bWEABnRhYmx1YQADA/4DAv4EBgEB  
AAID/P8A2U7wOA==  
YU1wXx8BAAAA0gAAAFkaAAAAAFgAAAAAAAEEAgAD//8AAQAAAABZAAAAABAAAAAUGFAAAAfdbV  
ng==  
'/*!*/;  
### UPDATE `db2_schema`.`tablea`  
### WHERE  
###   @1=1  
###   @2='A'  
###   @3=100|  
### SET  
###   @1=1  
###   @2='A'  
###   @3=133  
# at 6745  
#200927 10:12:17 server id 1  end_log_pos 6776 CRC32 0x4c5a23c6          Xid = 47  
COMMIT/*!*/;
```

Transactional Rules

- Log management rules ensure transactions implement write operations in a way that supports reliability
- A commit log record must be written synchronously (with a force operation)
- **Write-Ahead-Log**
 - Before-state part of the record must be written in the log before actually carrying out the corresponding operation on the database
 - In this way, actions can always be undone
- **Commit Rule**
 - After-state part of the record must be written in the log before carrying out the commit
 - In this way, actions can always be redone
- NOTE: as database writes are asynchronous different implementations are possible, with an impact on recovery

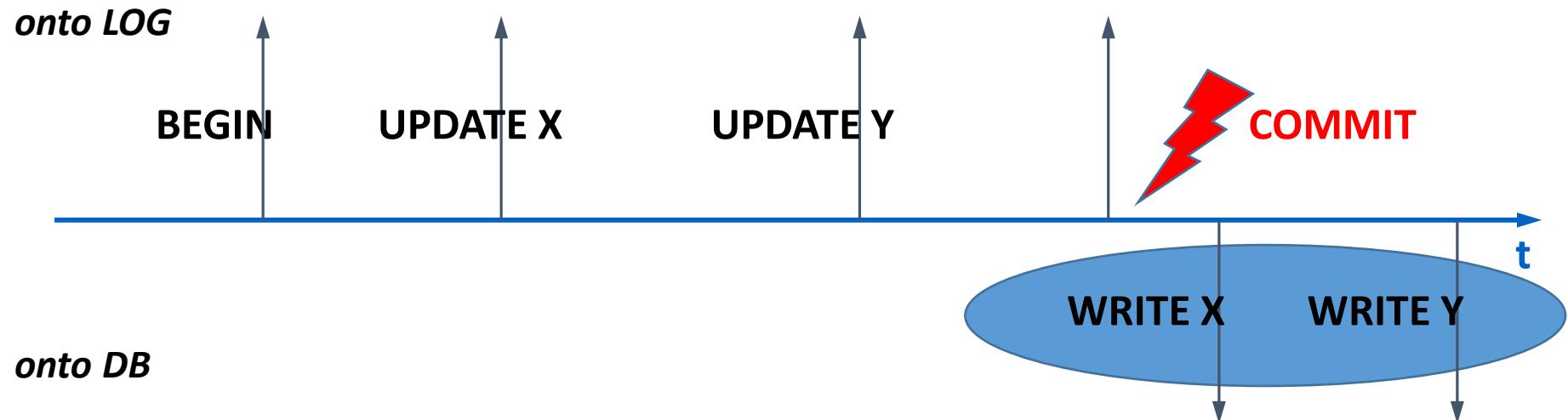
Writing onto Log and Database

- Writing onto the database before commit
 - Redo is not necessary, because the state of the database reflects the state of the log



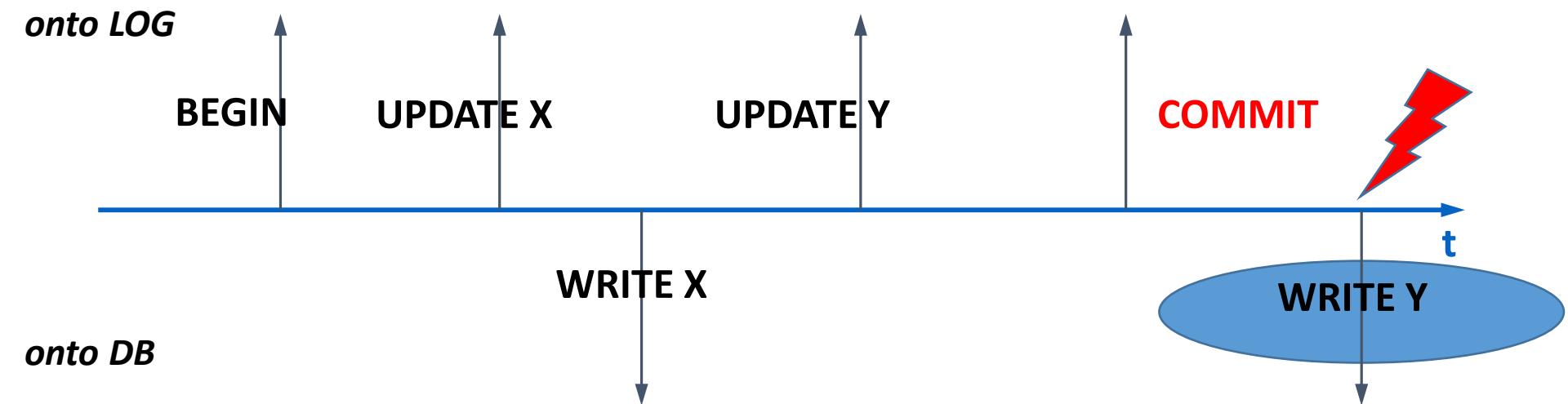
Writing onto Log and Database

- Writing onto the database after commit
 - Undo is not necessary
 - Does not require writing the before states of objects on the DB in order to abort



Writing onto Log and Database

- Writing onto the database in arbitrary points in time
 - Allows optimizing buffer management
 - Requires both undo and redo, in the general case

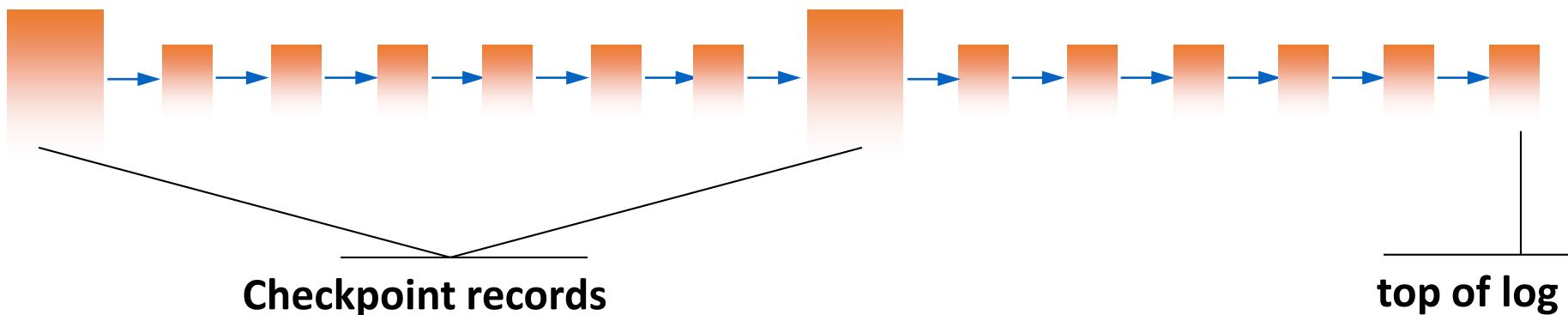


Recovery after a failure: types of failure

- **Soft failure**
 - Loss of the content of (part of) the main memory
 - Requires **warm restart**
 - Log is exploited to replay transaction operations
- **Hard failure**
 - Failure / loss of (part of) secondary memory devices
 - Requires **cold restart**
 - Dump is exploited to restore database content and log is exploited to replay transaction operations
- **Disaster**
 - Loss of stable memory (i.e., of the log and dump)
 - Not treated here

Checkpoint

- Performed periodically by the Reliability Manager to identify “consistent” time points
 - all transactions that committed their work flush their data from the buffer to the disk (effects are made persistent)
 - All active transactions (not committed yet) are recorded in the log
 - Aim: to record which transactions are still active at a given point in time (and, dually, to confirm that the others either did not start yet or have already finished)

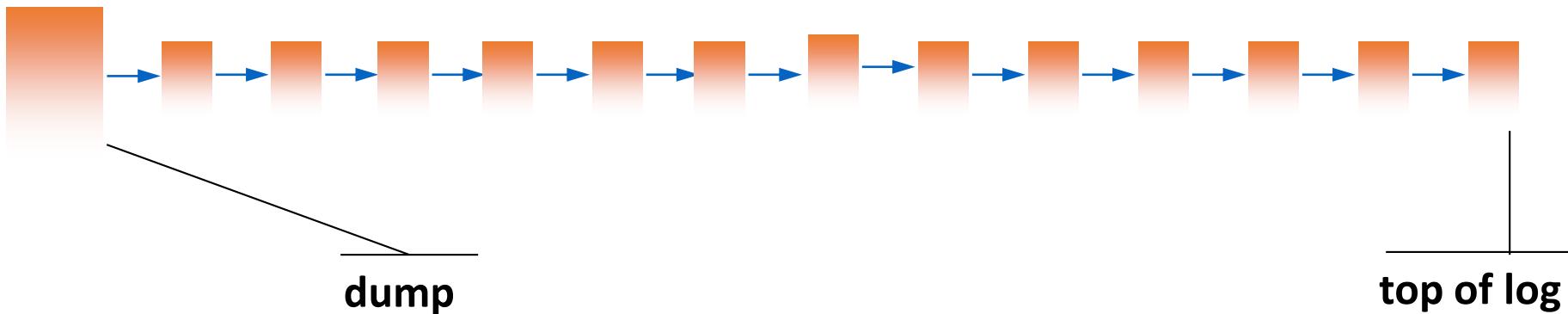


Checkpoint

- Several possibilities/variants – a simple option is as follows:
 - Acceptance of all commit and abort requests is suspended
 - All “dirty” buffer pages **modified by committed transactions** are transferred to mass storage (synchronously, via force)
 - First the log entries recording the operations on the page data are forced, if not yet done
 - Then the page is flushed
 - The identifiers of the transactions still in progress are recorded in the CKPT record in the log (synchronously, via force); also, no new transaction can start while this recording takes place
 - Then, acceptance of operations is resumed
- In this way, there is guarantee that:
 - for all **committed** transactions, data are now on mass storage
 - transactions that are “half-way” are listed in the checkpoint log record (in stable memory)

Dump

- A time point in which a complete backup copy of the database is created (typically at night or in week-ends)
- The availability of that copy (dump) is recorded in the log
- The content of the dump is stored in stable memory



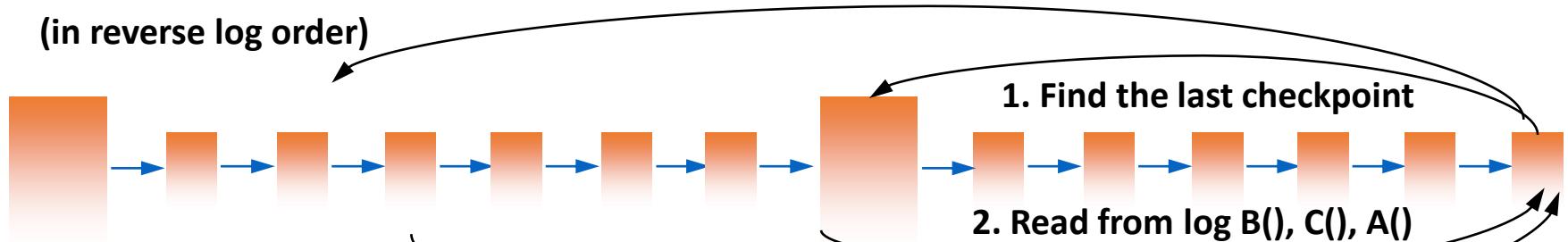
Warm Restart

- Loss of main memory buffer pages, not of table data on disk
- Requires “replaying” transactional operations and resolving in-doubt situations
- Log records are read starting from the last checkpoint (i.e., last stable point where all changes of committed transactions were flushed to disk)
 - CKPT record identifies the active transactions
- Active transactions are divided in two sets:
 - **UNDO** set: transactions to be undone
 - **REDO** set: transactions to be redone
- **UNDO and REDO** actions are executed

Warm Restart

- Find the most recent checkpoint
- Build the UNDO and REDO sets;
 - UNDO = active transactions@CKPT; REDO = empty;
 - For log records from CKPT to TOP
 - **IF B(Ti) then UNDO += Ti // started, may be undone**
 - **IF C(Tj) then UNDO -= Tj; REDO += Tj // ended, to redo**
- For all log records from TOP to 1st action of oldest T in UNDO
 - undo(Ti) where Ti is in UNDO
- For all log records from 1st action of oldest T in REDO to TOP
 - redo(Ti) where Ti is in REDO

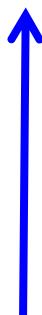
**3. Return to the 1st operation of the oldest active transaction while performing UNDO actions
(in reverse log order)**



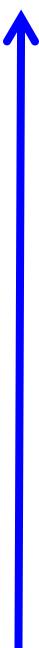
**4. Execution of REDO actions
(in log order) until the top of the log**

Example of Warm Restart

- B(T1)
 - B(T2)
 - U(**T1,O1,B1,A1**)
 - I(T1,O2,A2)
 - U(T2,O3,B3,A3)
 - B(T3)
 - U(T3,O4,B4,A4)
 - D(T3,O5,B5)
 - CKPT(T1,T2,T3)
 - C(T2)
 - B(T4)
 - U(T4,O6,B6,A6)
 - A(T4)
 - failure
- UNDO=(T1,T2,T3), REDO=()
 - UNDO=(T1, T3), REDO=(**T2**)
 - UNDO=(**T1,T3,T4**), REDO=(T2)



Example of Warm Restart

- B(T1) UNDO=(T1,T3,T4)
 - B(T2) REDO=(T2)
 - U(T1,O1,B1,A1)
 - I(T1,O2,A2)
 - U(T2,O3,B3,A3)
 - B(T3)
 - U(T3,O4,B4,A4)
 - D(T3,O5,B5)
 - CKPT(T1,T2,T3)
 - C(T2)
 - B(T4)
 - U(T4,O6,B6,A6)
 - A(T4)
 - failure
- 
- (5) O1 = B1
 - (4) delete(O2)
 - (6) O3 = A3
 - (3) O4 = B4
 - (2) Re-insert(O5=B5)
 - (1) O6 = B6
 - restart

Cold Restart

- Soft failure
 - Loss of the content of (part of) the main memory
 - Requires warm restart
- Hard failure
 - Failure / loss of (part of) secondary memory devices
 - Requires cold restart
- During cold restart
 - Data are restored starting from the last backup (dump)
 - The operations recorded onto the log until the failure time are executed (in log order)
 - Data on disk are restored in the status existing at the time of failure
 - A warm restart is then executed
 - Uncertain transactions are undone