CS-A1153

Databases for Data Science

Views, Indexes, Transactions and Triggers

May 11, 2021

Aalto University
School of Science

Instructors:
Prof. Nitin Sawhney
Prof. Barbara Keller

Acknowledgements

These slides are based in part on presentation materials created by **Kerttu Pollari-Malmi, Juha Puustjärvi and Lukas Ahrenberg** in previous years and on the course text book: **A First Course in Database Systems**, Third Edition. Pearson by Jeffery D. Ullman and Jennifer Widom.

Thanks to **Etna Lindy** & **Ville Vuorenmaa** for translating prior lecture slides for this course.

Views

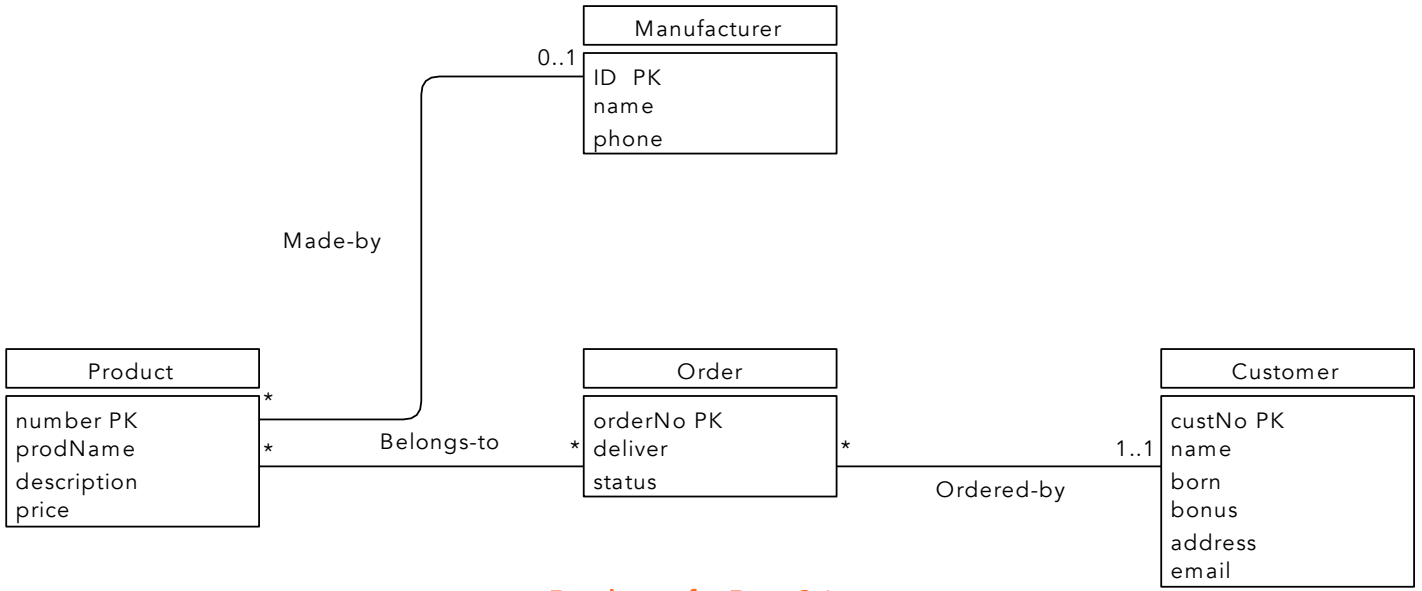
CS-A1153: Databases

Prof. Nitin Sawhney

Example Database

- Examples in this lecture use the database from prior lectures consisting of the following relations/tables:

  Customers(custNo, name, born, bonus, address, email)
  Products(number, prodName, description, price, manufID)
  Manufacturers(ID, manufName, phone)
  Orders(orderNo, deliver, status, custNo)
  BelongsTo(orderNo, productNo, count)



Views

- While we can create tables that are persistently stored in a database, we can also define **views** created as "virtual tables" that are temporarily created by query expressions.
- A view can be defined with the command:

  CREATE VIEW name **AS** definition;

  where name is the name of the view we are defining, and definition is some SQL query.
- Example: let's create a view, that contains only the products with the manufacturer "Samsung" from *Products*:

```
CREATE VIEW SamsungProducts AS
    SELECT number, prodName, description, price, manufID
    FROM Products, Manufacturers
    WHERE manufName = 'Samsung' AND manufID = ID;
```

Views: Example

- Example: let's create a view with the number and name of a product combined with the name of the manufacturer:

```
CREATE VIEW ProdManuf AS
    SELECT number, prodName, manufName
    FROM Products, Manufacturers
    WHERE manufID = ID;
```

- We can target queries on views in a same manner as with ordinary tables. We can use both ordinary tables and views in the same query.
- Example: search for order numbers of orders that contain products from Samsung *(using a view created from Products):*

```
SELECT DISTINCT orderNo
FROM SamsungProducts, BelongsTo
WHERE number = productNo;
```

Using Views

- With a view we can simplify the queries we want to write, as part of the query is in a sense hidden behind the definition of the view.

- Using a view doesn't make the query more efficient, as the view is created again (the query defining the view will be executed) whenever the view is used in a query.

- In addition, we can use views to manage the permissions of the users of the database. We can define, that a user can use a restricted set of views, but not access the original tables.

Renaming Attributes

- While defining a view, we may rename the attributes inside parenthesis after the name of the view.
- Example:

```
CREATE VIEW ProdManuf(productNumber,
productName, ManufacturerName)
SELECT number, prodName, manufName
FROM Products, Manufacturers
WHERE manufID = ID;
```

- This view is identical to the view in the earlier example, but here the attributes have different names.

Note: This does not seem to work in SQLite when tested, even though it should, based on the documentation.

Modifying Views

- We can remove a view with the command DROP VIEW, for example:

  DROP VIEW ProdManuf;

  This command only deletes the definition of the view, not the tables used to create the view.

- It's possible to write SQL queries, that dynamically update the view (insert, delete or update tuples of the view), though the SQL rules for creating *Updatable Views* are complex. However, once created they can be queried and modified like regular views.
- Views can be *materialized*, which means storing the view and updating it whenever it's necessary. In this course, we won't cover views as thoroughly, but in the next lecture we will cover indexes which are a form of materialized views.

Indexes

CS-A1153: Databases

Prof. Nitin Sawhney

INDEXES

- What is an index
- Why DB indexes are needed
- What indexes to create
- U&W 8:3-8:4
- Additional Example by Kerttu Pollari-Malmi (similar to 8:4, ex 14)

THE PROBLEM

- Databases are typically large
- The data needs to be stored somewhere, *and in some order* (often 'random')
- The tuples accessed by some query could require the DB system to go over all data on disk just to find those which matches a very narrow condition

INDEXES - A SOLUTION

An index on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A

- Indexing a way to build up information of where some information is located in the DB table
- The attributes of the index is called the *index key* (these need not be the same as the keys of the relation)
- Note that the index itself needs to be stored
- And, crucially, *the index needs to be updated*
- Often implemented using B-trees

EXAMPLE

```
SELECT * FROM Movies
       WHERE studioName = 'Disney'
       AND year = 1990;
```

- No index means that all movies needs to be searched
- Index on e.g. `year` means DB can quickly narrow down search
- Indexes also products and joins where a table might need to be traversed multiple times

CREATING AN INDEX IN SQL

Creation of indexes not part of SQL standard, but commonly done as

```
CREATE INDEX <index_name> ON <table_and_attributes>;
```

For example:

```
CREATE INDEX KeyIndex ON Movies(title,year);
```

WHICH INDEXES SHOULD BE CREATED?

- How are the tuples of a relation spread over the storage medium?
  - Usually spread over several 'pages' (sequential blocks)
- What are common queries? When does it make sense to create an index?
- In practice there is also a cost associated with reading and updating the index

INDEXES ON RELATION KEYS

- Queries involving the relation key are common, and
- An index on a key yields a single page (because the key is unique)

INDEXES ON 'ALMOST' KEYS

Queries involving 'almost' keys can result in few page hits

```
SELECT * FROM Movies
    WHERE title = 'King Kong';
```

- `title` is not a key of **`Movies`**, but
  - In general there are only a handful of movies with the same name
  - So, few pages to read (compared to the total number of the DB)

INDEXES ON CLUSTERED ATTRIBUTES

If we know that the attribute is *clustered* on a few storage pages, it can also make sense to create an index.

```
SELECT * FROM Movies
       WHERE year = 1990;
```

- `year` is not a key of Movies, but
  - if the movie table should happen to be clustered on year on the storage medium it still makes sense to create an index
  - instead of searching all pages for the right year, we would look up the correct page(s) in the index

Transactions

CS-A1153: Databases

Prof. Barbara Keller

Aalto University
School of Science

Beginner's Python for Engineers

Title: Beginner's Python for Engineers

Credits: 2 ETCS

Start: 18.th of May

Duration: 6 weeks (18.05.2021 - 22.06.2021)

Weboodi: https://oodi.aalto.fi/a/opettaptied.jsp?OpetTap=1149002605&html=1

myCourses: https://mycourses.aalto.fi/course/view.php?id=31059

Teacher in charge: Barbara Keller

Target audience:
Everyone who wants to deepen their programming skills by using python to solve engineering tasks and analyze data
Prerequisites: Basic programming skills similar to the skills from CS-A1113 Basics in Programming Y1 (while and for-loop / if-then-else)

Some selected **learning outcomes**:
- From data given in a CSV file, you are able to tell the min, max and average
- Given a data set you are able to clean your data for further processing
- You can explain what regression is and apply it
- You will be able to present your scientific findings and your data in an appropriate way

Calculations Yeah ☺

Example from the book (thanks to Lukas Ahrenberg – slides mainly copied)

Similar example with different numbers to check out from Kerttu Pollari-Malmi:
http://www.cse.hut.fi/fi/opinnot/CS-A1150/K2020/luennot/index-example.pdf

Example 14 (U&W 8:4.3)

StarsIn(movieTitle, movieYear, StarName)

movieTitle	year	starName
Harry Potter Philosopher Stone	2001	Emma Watson
Harry Potter Chamber of Secrets	2002	Emma Watson
Spider-Man No Way Home	2021	Tom Holland
Harry Potter Philosopher Stone	2001	Rupert Grint
Spider-Man	2002	Tobey Maguire
Harry Potter Half Blood Prince	2009	Daniel Radcliffe
Harry Potter Philosopher Stone	2001	Rupert Grint
Spider-Man	1977	Nicholas Hammond

Where shall we put an index?

What happens most often with this table?
(Design: The most used feature the easisest to find ;))

$Q_1(s)$: In which movie and when played s?

$Q_2(m, y)$: Which star played in movie m in year y?

$I(t, y, s)$: New movies → update the table

Assumptions

- StarsIn occupies 10 pages
- Ø a star has appeared in 3 movies
- Ø a movie has 3 stars
- An index fits on one page
- Inserting is easy (1 read / 1 write)

$Q_1(s)$: In which movie and when played s?
$Q_2(m, y)$: Which star played in movie m in year y?
$I(t, y, s)$: New movies → update the table

How do we calculate it?

- Fraction p_1 is of type Q_1
- Fraction p_2 is of type Q_2
- Fraction $1 - p_1 - p_2$ is of type I

Cost: $C_{Q_1} p_1 + C_{Q_2} p_2 + \mathrm{C_I}(1 - \mathrm{p_1} - \mathrm{p_2})$

Lets compare the costs: No Index

Q_1: 10 (Need to scan the whole table)

Q_2: 10 (Need to scan the whole table)

I: 2 (One access to read the page and one to write it back modified)

Average:

Lets compare the costs: No Index

Q_1: 10 (Need to scan the whole table)

Q_2: 10 (Need to scan the whole table)

I: 2 (One access to read the page and one to write it back modified)

Costs: $C_{Q_1} p_1 + C_{Q_2} p_2 + \mathrm{C_I}(1 - \mathrm{p_1} - \mathrm{p_2})$

Average: $10p_1 + 10p_2 + 2(1 - p_1 - p_2) = 2 + 8p_1 + 8p_2$

Lets compare the costs: Index on starName

Q_1: 4 (One to read the index, and 3 pages in Ø)

Q_2: 10 (Need to scan the whole table)

I: 4 (2 to write new data + 2 to update index)

Costs: $C_{Q_1}p_1 + C_{Q_2}p_2 + \mathrm{C_I}(1 - \mathrm{p_1} - \mathrm{p_2})$

Average: $4p_1 + 10p_2 + 4(1 - p_1 - p_2) = 4 + 6p_2$

Lets compare the costs: Index on (movieTitle,movieYear)

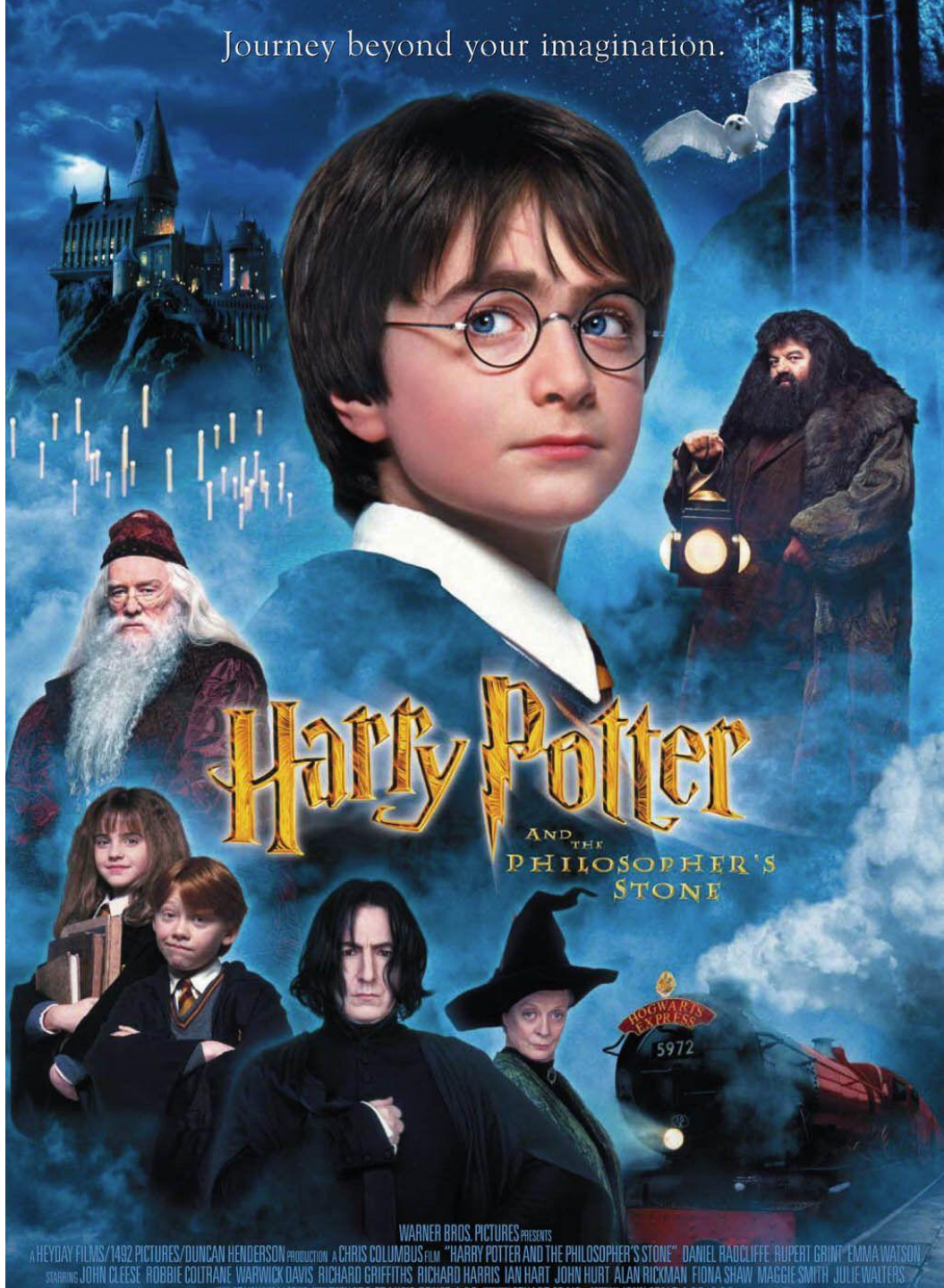
Q_1: 10 (Need to scan the whole table)

Q_2: 4 (One to read the index, and 3 pages in ∅)

I: 4 (2 to write new data + 2 to update index)

Costs: $C_{Q_1}p_1 + C_{Q_2}p_2 + \mathrm{C_I}(1 - \mathrm{p_1} - \mathrm{p_2})$

Average: $10p_1 + 4p_2 + 4(1 - p_1 - p_2) = 4 + 6p_1$



Lets compare the costs: Index on both

Q_1: 4 (One to read the index, and 3 pages in Ø)

Q_2: 4 (One to read the index, and 3 pages in Ø)

I: 6 (2 to write new data + 2x2 to update each indices)

Average:?

Lets compare the costs: Index on both

Q_1: 4 (Need to scan the whole table)

Q_2: 4 (One to read the index, and 3 pages in ∅)

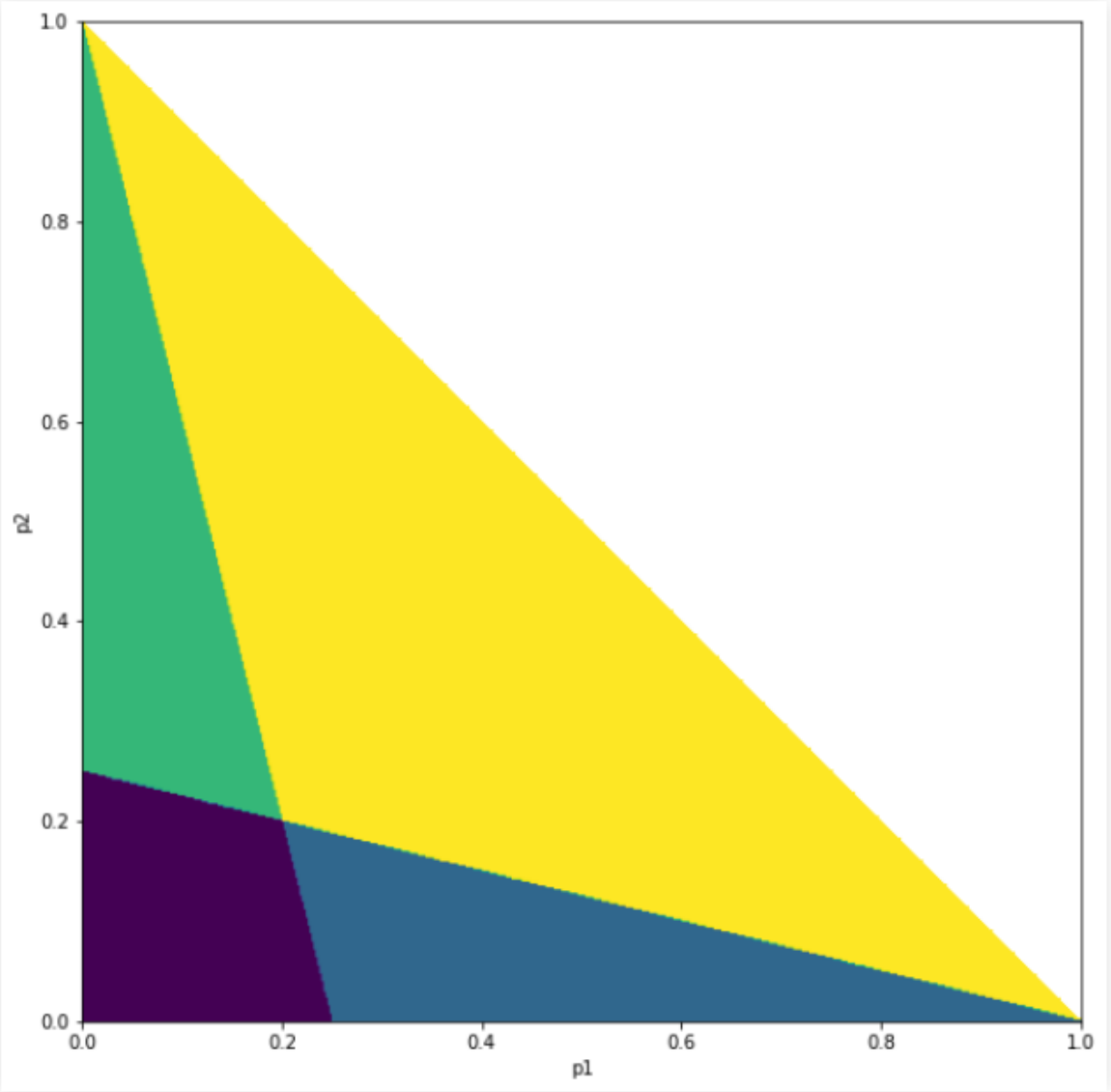
I: 6 (2 to write new data + 2 to update index)

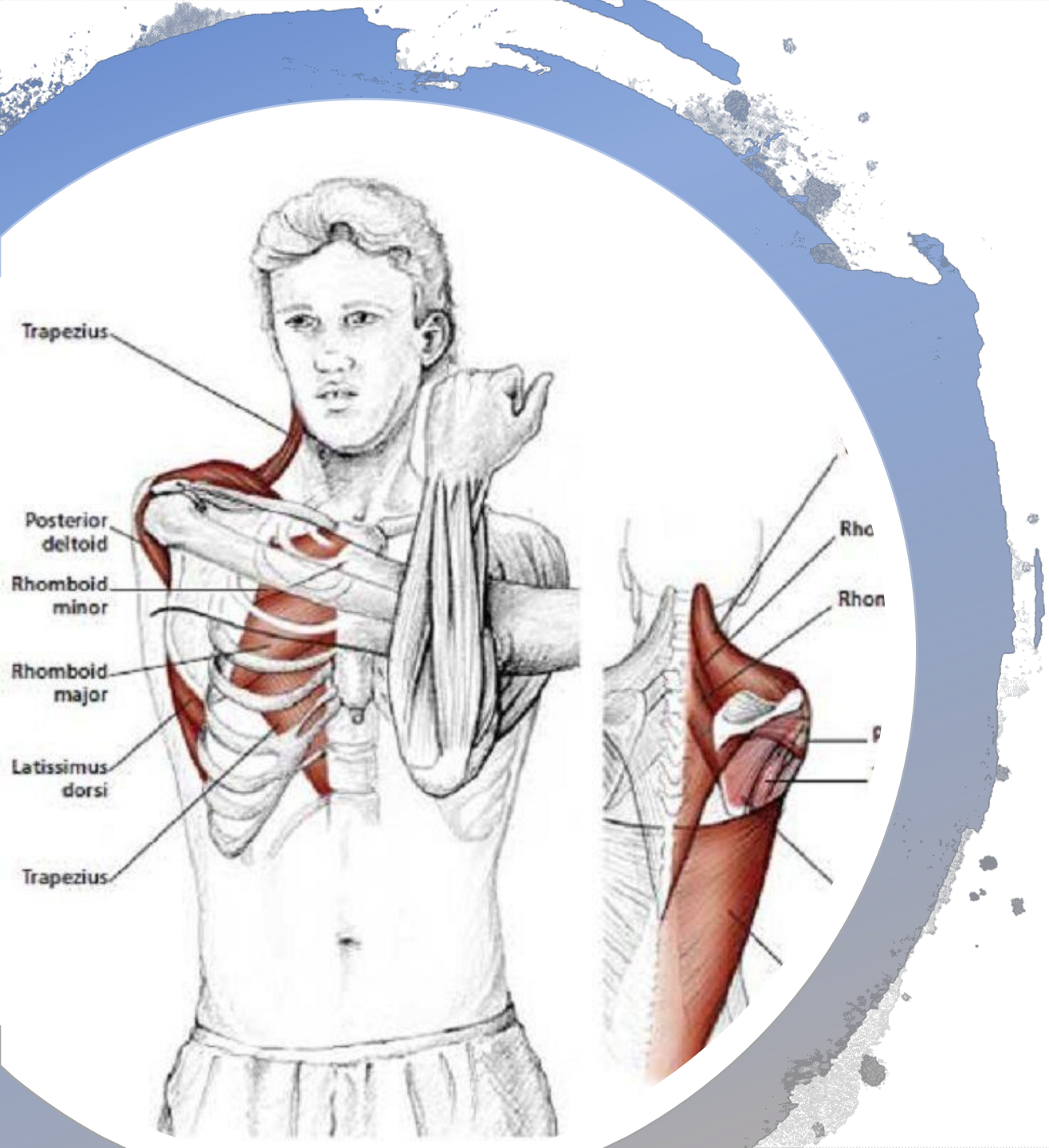
Costs: $C_{Q_1} p_1 + C_{Q_2} p_2 + \mathrm{C_I}(1 - \mathrm{p_1} - \mathrm{p_2})$

Average: $4p_1 + 4p_2 + 10(1 - p_1 - p_2) = 6 - 2p_1 - 2p_2$

Comparison

No index	Star index	Movie Index	Both indices
$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$





Break:
Move your Shoulders

Transactions and ACID

Thanks to Mikolaj Wojnicki

Andreas Reuter

Intended Learning Outcomes

After this session you can:

- explain in your own words what ACID stands for
- describe what a transaction is and why it is an important concept
- figure out what can happen in a database given some transactions

DB in the Wild

I pay you back the DB you bought for me (100)

	Barbara	Sami
T1:	350	450

DB in the Wild

Does Barbara have enough money?
Money > 100

	Barbara	Sami
T1:	350 ✓	450

DB in the Wild

Money –= 100

	Barbara	Sami
T1:	350	450
T2:	250	

DB in the Wild

Money += 100

	Barbara	Sami
T1:	350	450
T2:	250	

DB in the Wild

Money += 100

	Barbara	Sami
T1:	350	450
T2:	250	550

DB in the Wild

Money –= 100

Fail

	Barbara	Sami
T1:	350	450
T2:	250	

Transactions

A transaction is a collection of one or more operations on the database that must be executed atomically: That is, either all operations are performed or none are
U&W: 1:24,6:6

START TRANSACTION
<statements>
COMMIT;

DB in the Wild

I pay you back the DB you bought for me (100)

I pay you back the DB you bought for me (100)

	Barbara	Sami	Nitin
T1:	350	450	550
T2:			

DB in the Wild

Does Barbara have enough money? Money > 100

Does Nitin have enough money? Money > 100

	Barbara	Sami	Nitin
T1:	350	450	550
T2:	350	450	550

DB in the Wild

Money -= 100

Money -=100

	Barbara	Sami	Nitin
T1:	350	450	550
T2:	350	450	550
T3:	250	450	

DB in the Wild

Money -= 100

Money -=100

	Barbara	Sami	Nitin
T1:	350	450	550
T2:	350	450	550
T3:	250	450	450

DB in the Wild

Money = 450

Money = 450

	Barbara	Sami	Nitin
T1:	[illegible]0	450	5[illegible]
T2:	350	450	550
T3:	250	450	450
T4:	250	450	450

DB in the Wild

Money += 100

Money = 450

	Barbara	Sami	Nitin
T1:	250	450	550
T2:	350	450	550
T3:	250	450	450
T4:	250	450	450
T5:	250	550	

DB in the Wild

Money += 100

Money += 100

Concurrency

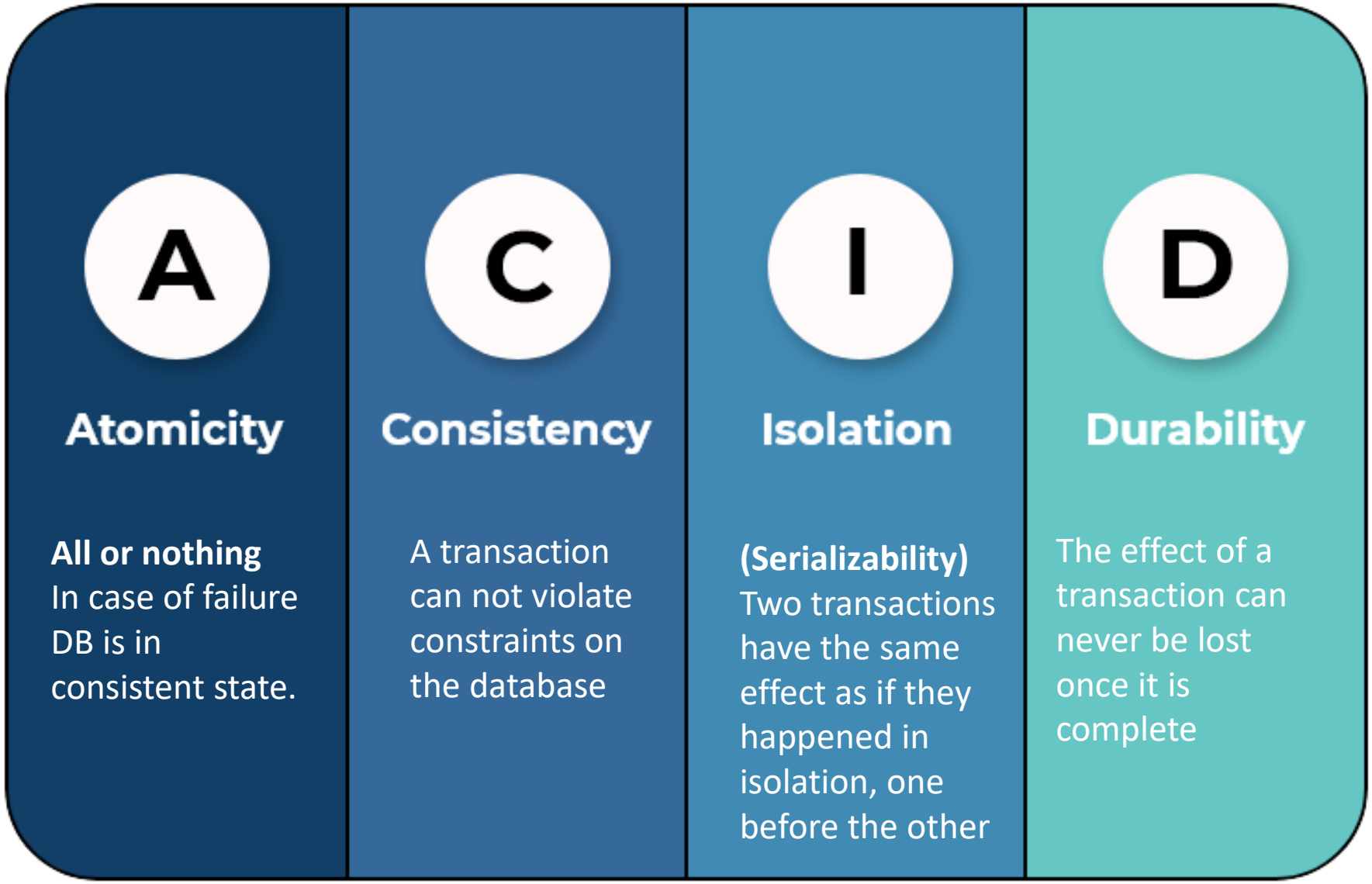
		Sami	Nitin
T1:		450	550
T2:			
T3:			
T4:	250		
T5:	250	550	450
T6:	250	550	450

Solutions

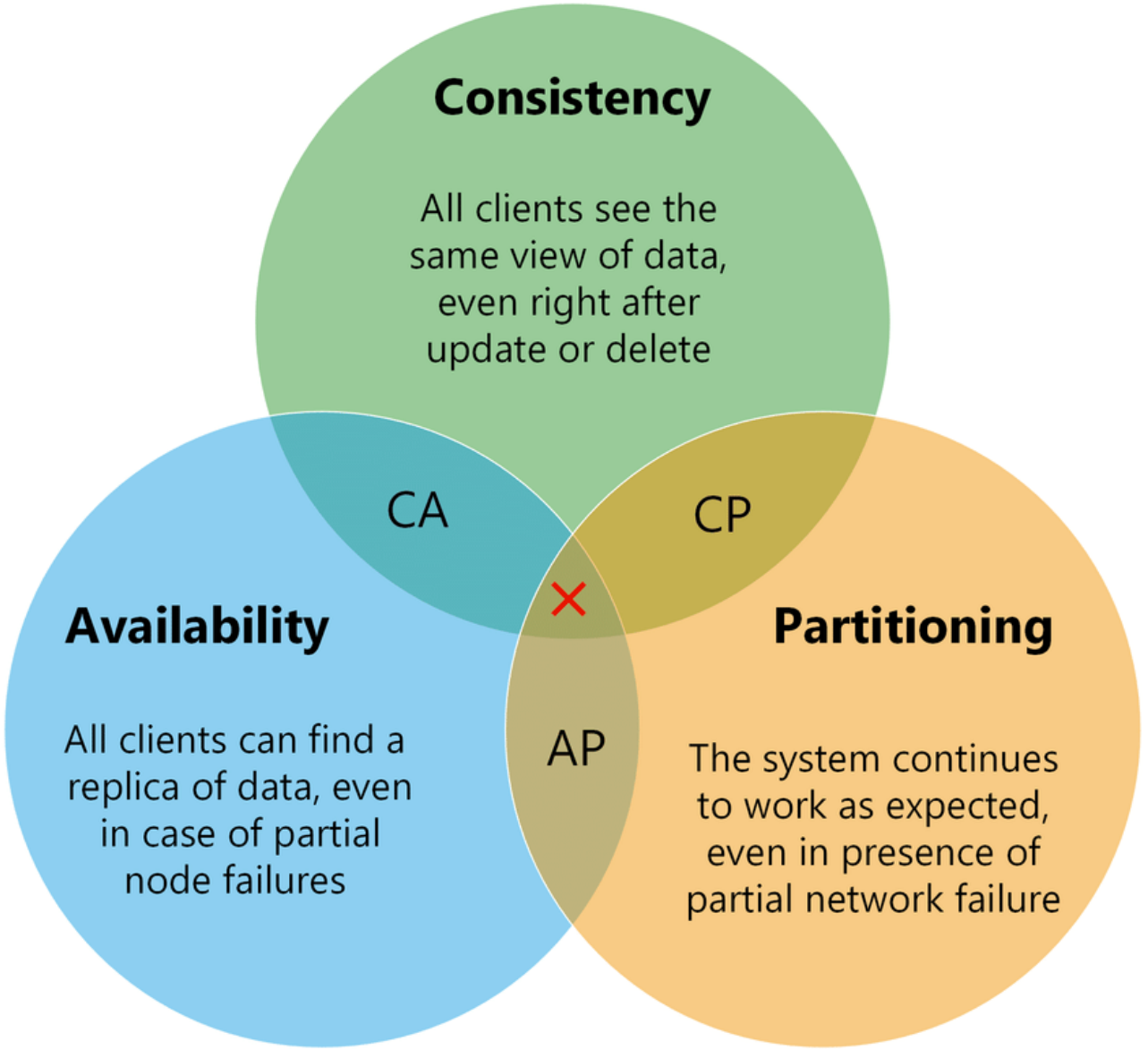
- Logging
  We know what happened and we can ”clean up” if something goes wrong (e.g. Rollback)
- Concurrency Control
  We know what can happen at the same time and what not: We use locks to ensure that we do not run into problems
- Deadlock resolution
  Who goes first?

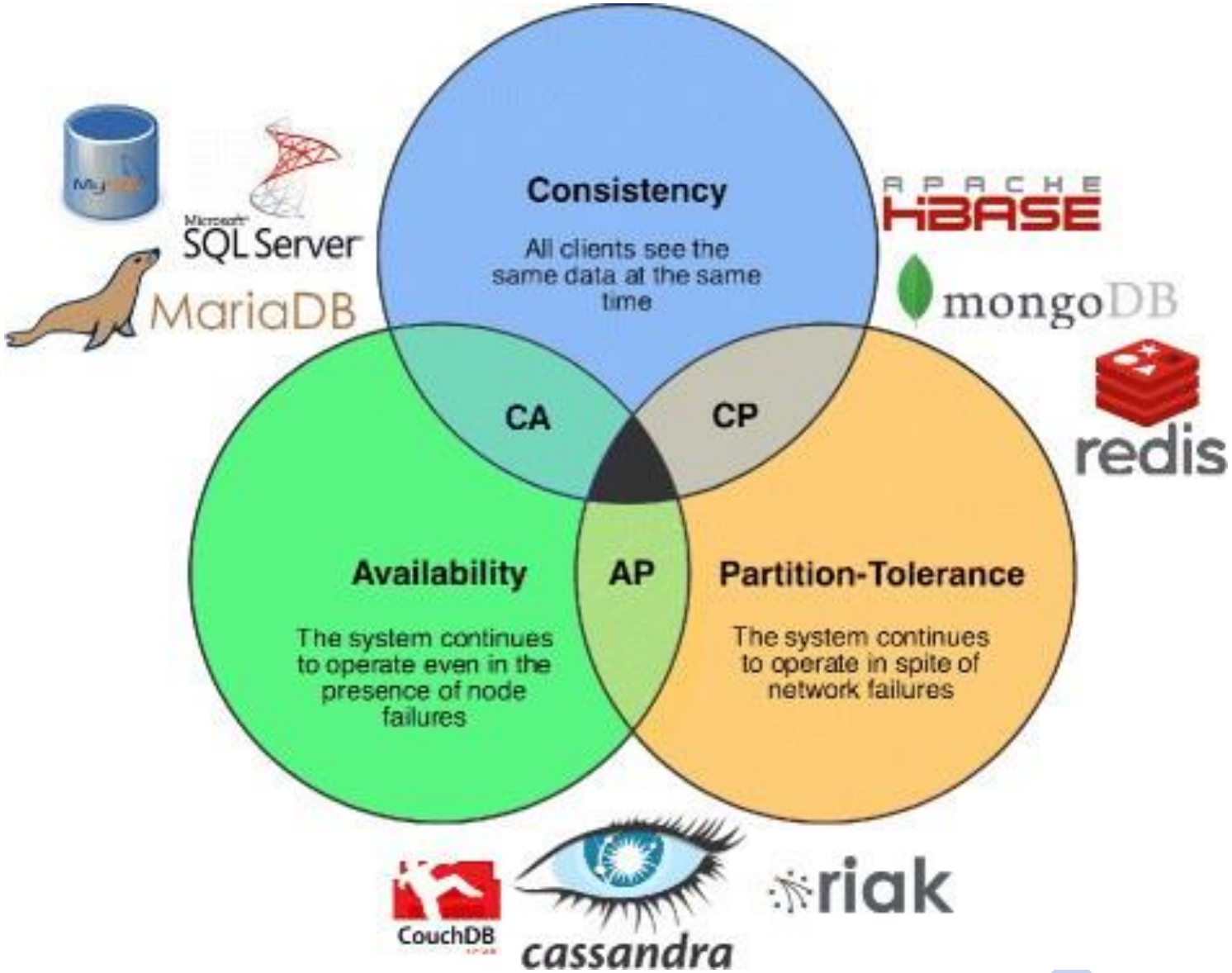
Solutions

- Logging
  We know what happened and we can ”clean up” if something goes wrong (e.g. Rollback)
- Concurrency Control
  We know what can happen at the same time and what not: We use locks to ensure that we do not run into problems
- Deadlock resolution
  Who goes first?


A
Atomicity
All or nothing
In case of failure DB is in consistent state.
C
Consistency
A transaction can not violate constraints on the database
I
Isolation
(Serializability)
Two transactions have the same effect as if they happened in isolation, one before the other
D
Durability
The effect of a transaction can never be lost once it is complete

CAP-Theorem


Consistency
All clients see the same view of data, even right after update or delete
CA
CP
Availability
All clients can find a replica of data, even in case of partial node failures
AP
Partitioning
The system continues to work as expected, even in presence of partial network failure


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MariaDB
SQL Server
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

Isolation Levels SQL

The isolation level of a transaction specifies what that particular transaction may see

- **SERIALIZABLE:**
  No other transaction may write to any of the data fields this transaction is working with until it finishes

- **REPEATABLE READ**
  This transaction can read *committed data* by other transactions which may execute simultaneously and *repeated reads* within this transaction must be consistent
- **READ COMMITED**
  This transaction can read committed data by other transactions which may execute simultaneously, but repeated reads are not necessarily consistent
- **READ UNCOMMITTED**
  This transacxtion can read dirty data not yet committed by other transactions

Triggers

CS-A1153: Databases

Prof. Nitin Sawhney

TRIGGERS - LEARNING

- What are triggers
- Why/When to use them
- How?
  - Operations
  - Before/after operation
- Difference between row-level and statement-level triggers

U&W 7:5

TRIGGERS

- A **Trigger** sets up an action for when some condition becomes true (*when something changes*)
- An **event** (insert, delete, update, …) fulfilling some **condition** triggers some further **action**
- The action can happen once for the triggering statement, or once for each tuple resulting from the triggering statement

TRIGGERS IN SQL

Created with statement `CREATE TRIGGER` (see examples)

Two types in SQL standard:

row-level trigger

Once for each modified tuple

statement-level trigger

Once for all the tuples changed in one SQL statement (not supported by SQLite)

EXAMPLE ROW-LEVEL TRIGGER

PC(model,speed,ram,hd,price)

An update to PC price may lower it at most €100:

(This syntax not supported in SQLite)

```
CREATE TRIGGER PriceFloor
AFTER UPDATE OF price ON PC
REFERENCING
        OLD ROW AS OldTuple,
        NEW ROW AS NewTuple
FOR EACH ROW
WHEN (NewTuple.price < OldTuple.price - 100)
     UPDATE PC SET price = OldTuple.price - 100
     WHERE model = NewTuple.model
     ;
```

ROW-LEVEL TRIGGER CONT.

PC(model,speed,ram,hd,price)

An update to PC price may lower it at most €100:

(SQLite syntax)

```
CREATE TRIGGER PriceFloor
AFTER UPDATE OF price ON PC
FOR EACH ROW
WHEN (NEW.price < OLD.price - 100)
BEGIN
      UPDATE PC SET price = OLD.price - 100
      WHERE model = NEW.model
      ;
END;
```

ROW-LEVEL TRIGGER CONT.

```
UPDATE PC SET price = 0.8*price;
```

Before update:

model	speed	ram	hd	price
1001	2.7	8192	1000	999
1002	1.6	4096	500	299
1003	3.0	4096	500	499
1004	1.6	8192	1000	1249
1005	1.4	4096	500	499
1006	3.2	8192	1000	799
1007	3.5	8192	1000	749
1008	3.2	8192	1000	499
1009	2.8	8192	1000	1249
1010	3.2	8192	1000	2099
1011	3.3	8192	700	999
1012	1.4	2048	16	219
1013	1.6	4096	500	329

After update:

model	speed	ram	hd	price
1001	2.7	8192	1000	899
1002	1.6	4096	500	239.2
1003	3.0	4096	500	399.2
1004	1.6	8192	1000	1149
1005	1.4	4096	500	399.2
1006	3.2	8192	1000	699
1007	3.5	8192	1000	649
1008	3.2	8192	1000	399.2
1009	2.8	8192	1000	1149
1010	3.2	8192	1000	1999
1011	3.3	8192	700	899
1012	1.4	2048	16	175.2
1013	1.6	4096	500	263.2

EXAMPLE - STATEMENT-LEVEL TRIGGER

Laptop(model,speed,ram,hd,screen,price)

No price update can result in an average laptop price of more than €2000.

Select level triggers not supported in SQLite.

```
CREATE TRIGGER AvgPriceTrigger
AFTER UPDATE OF price ON Laptop
REFERENCING
        OLD TABLE AS OldTable,
        NEW TABLE AS NewTable
FOR EACH STATEMENT
WHEN (2000 < (SELECT AVG(price) FROM Laptop))
BEGIN
        DELETE FROM Laptop
        WHERE (model,speed,ram,hd,screen,price) IN NewTable;
        INSERT INTO Laptop (SELECT * FROM OldTable);
END;
```

CREATE TRIGGER - SOME SYNTAX

At what point

BEFORE / AFTER

Operation

DELETE , INSERT , UPDATE

WHEN

Is optional, without it trigger always on operation

FOR EACH STATEMENT

is default in SQL standard

EXAMPLE - BEFORE INSERT

Check before a new PC is inserted, if price is NULL use default value of €150:

(SQLite)

```
CREATE TRIGGER FixPriceOnInsert
BEFORE INSERT ON PC
FOR EACH ROW
    WHEN (NEW.price IS NULL)
    BEGIN
        INSERT INTO PC VALUES (NEW.model, NEW.speed,
                NEW.ram, NEW.hd, 150);
        SELECT RAISE (IGNORE);
    END
    ;
```

EXAMPLE - AFTER DELETE

When a PC model is deleted, increase the price of other PC models with the same amount of ram by 20%, and of Laptop models with same amount of ram by 10%.

(SQLite)

```
CREATE TRIGGER PriceHikeOnDelete
AFTER DELETE ON PC
FOR EACH ROW
    BEGIN
    UPDATE PC SET price = price * 1.2
           WHERE ram = OLD.ram;
    UPDATE Laptop SET price = price * 1.1
           WHERE ram = OLD.ram;
    END
    ;
```