Databases!

Introduction

DDL	Data-Definition Language — CREATE, DROP, ALTER
DML	Data-Manipulation Language — INSERT, UPDATE, DELETE, SELECT
Relation schema	Name of table + attributes + types
instance	Records in the table at a certain point in time
Database	a collection of Relations
Key	duplication is forbidden!
	Primary key can f.ex. be I.D. Possibly Many keys, but 1PK
	Foreign key refers to PK in other schema
	Super key when a collection of columns are a key
Primitive types	INT / INTEGER ← Round numbers
all can be NULL	REAL / FLOAT ← Decimal numbers
	CHAR(n) / VARCHAR(n) ← strings → fixed/variable length
	DATE ← dates

Basic Queries

SELECT * FROM <relation> WHERE <condition>

LIKE matches string, '%' - any string '_' - any char
Distinct no duplicates

GROUP BY <columns> HAVING <condition>
↑ needed when aggregating
HAVING ↑ only consider grouped records where...
ORDER BY <columns>
↑ only for pretty outputs

<Relation 1> JOIN <Relation 2> ON <col 1> = <col 2>

SUM, AVG, COUNT, MIN, MAX, ...
↑ aggregate functions

Complex Queries

Division — which x has ALL possible y

solution: group by x having count(y) = (subquery #Distinct y's)

Joins

INNER JOIN — A JOIN B ON...

NATURAL JOIN — i am dangerous — joins on all columns with same name

CROSS JOIN — rarely useful — no removed entries → join on 1=1

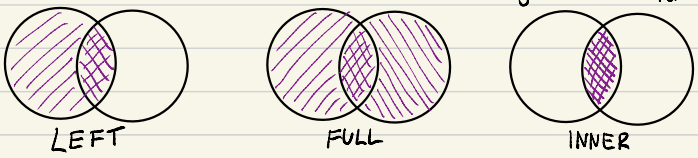
SELF JOIN — f.ex. self join with 'where c1.name < c2.name' will yield all pairs of names where join-cond are equal

OUTER JOIN ← can be LEFT, RIGHT, FULL

LEFT: left is main

RIGHT: right is main

FULL: nothing removed



Set Operation — <Query 1> UNION / INTERSECT / EXCEPT <Query 2>

same columns

UNION — we take 2 results, and plaster together

INTERSECT — we take 2 results, and return common rows

EXCEPT — we return the ones from Q1 that are not in Q2

Duplicates are removed by default...

↳ you want 'em? solution: ALL

UNION ALL — when Q1 & Q2 are disjoint!

EXISTS(<subquery>) → TRUE if subquery non-empty

in these subqueries, we sometimes use attributes from main-query

Views & Subqueries

Division

find x that have all y
(count(*) from y) ← subquery!
Group by x Having count(distinct y) = ( )

Subqueries

SELECT - FROM - WHERE as conditions / tables
Where x = (subquery) ← single tuple
Where x IN (subquery) ← multi-value
Where EXISTS (subquery) ← returns anything?
↑ only this can include outer tables

Where x >= ANY (subquery) ← multi-value, True for at least 1
x >= ALL (subquery) ← multi-value, True for all

Views

CREATE VIEW <NAME>(<columns>) AS
<Query>;

SQL Programming

Functions — Callable, runs queries/commands + logic (if, & loops)

How to use — From Java via JDBC OR SQL-script
May be faster than executing on a client...

Syntax:

```
DROP FUNCTION IF EXISTS function_name( );
CREATE FUNCTION function_name(IN var dtype)   ← initiate
RETURNS dtype ← what does it return?
AS $$ ← start of function text
DECLARE var dtype; ← declare local variables
BEGIN ← Begin SQL
    SELECT something INTO var   ← put in local variable
    FROM something ... ;        } Query
    RETURN var; ← Return value
END $$ LANGUAGE plpgsql;
   ↑ End SQL   ↑ End function text   ↑ Specify language
```

Triggers — Helps maintain integrity of data
Trigger: Insert/Update/Delete
Trigger-data: NEW.attribute AND/OR OLD.-"-
TG_OP variable containing operation type

1. Drop function & Trigger if exists
2. Create function 'returns TRIGGER', RETURN NEW
3. Create Trigger ;)

```
CREATE TRIGGER trigger_name
AFTER/BEFORE INSERT OR UPDATE OR DELETE
ON Relation
FOR EACH ROW EXECUTE PROCEDURE function
```

Transactions

When we want to make sure to do all the neccessary code in 1 block!
all or nothing execution!

```
BEGIN;
COMMIT;
ROLLBACK;
```

if we make it to commit, then it is done

Rollback then can undo what we just did. nothing is changed!

```
SAVEPOINT <name>;
ROLLBACK TO SAVEPOINT <name>;
```

SQL-Injections on web services can delete your ENTIRE database

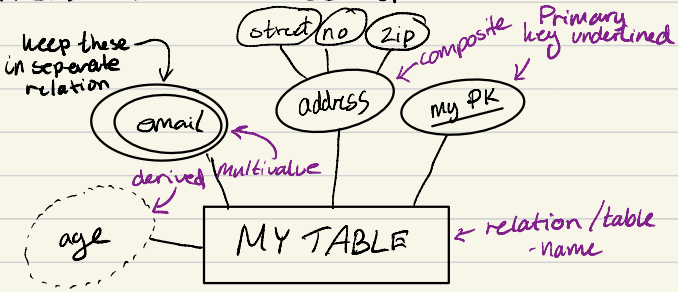
Entity Relationships

ER-diagrams

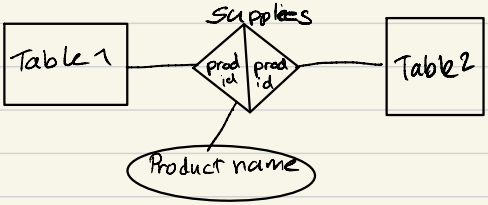
Why? Waay more interpretable!
Gives an overview of the DB schema

Entity Type Corresponds to a record/instance relation name
F.ex 'Students' with 'John', 'CSE305' in it

Attribute Type Corresponds to an aspect attribute names
F.ex. name & course for above



Relations Relate entities with roles



Cardinality Min: 0 or 1 Max: 1, N/M/L/*/...

Review

Cardinality, 0...N

What is ~~total~~ and disjoint

& Generalization/Complexity

Star-schema ↔

Practical Normalization

why? Improving the design & runtime

Redundancy Issues f.ex. zip ↔ city (have one, can deduce other)

Functional Dependency (FD) I.e. we don't need both columns

Solution **Decomposition:** new table that has zip → city mapping

How?

1. create new table
2. insert mapping from prior table
3. remove column from prior

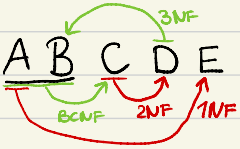
Normal Forms

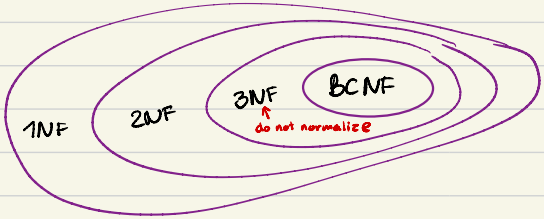
NF Before we decompose; i.e. redundant cols.

BCNF After we decompose; i.e. redundancy removed

Boyc-Codd Normal Form

every attribute is dependent ONLY on the whole key





1NF Part of key refers to other entity $\underline{AB}\,CD \rightarrow \underline{AB}C\ \underline{A}D$

2NF non-key entity refers to other entity $\underline{AB}\,CD \rightarrow \underline{AB}C\ \underline{C}D$

3NF non-key entity refers to key entity $\underline{AB}CD$

Decompose 2NF before 1NF

Storage Hierarchy, Multicore OS

cache	16 × 8 B	Core		
	32,256 KB 8-10 MB	L1, L2, L3	RAM (Random Access Memory)	hardware managed
Main Memory	16-64 GB	DRAM	~ ns	
Local Disk	1-2 TB	+ parallelism + flash chips; like 'bladespiller' SSD (Solid State Disk)	~ µs	software managed
Remote Disk		Hard Disk	~ ms	
Archival	8 TB	Tape	~ s	

Sequential access is faster than random access

When fetching:

- from Disk: 4 KB Blocks
- from Main Memory: 64 B Blocks
- from caches: 4-8 B Blocks

!Improve Locality for frequently accessed data

Parallelism / Multi-core

CPU Central Processing Unit

Multi-core several cores grouped } more parallelism :)

Multi-socket several layers grouped }

Multi-core parallelism has to be actively exploited

Operating Systems

Resource Management → Hardware is limited :o

Indexing

Runtime Analysis : EXPLAIN ANALYZE <QUERY>

CREATE INDEX

CREATE INDEX <name> ON <table> (<column>)

Goal: Avoid reading in random order.

Return 80%<

Full Table Scan: we read through EVERY record and return relevant tuples

Point Queries

<column> = <value> : easy if sorted by <column>

Range Queries

<column> BETWEEN <value1> and <value2> : easy if sorted by <column>

Solution: Indexes a sorted hashmap of a columns values

Built as hashing or Tree Search

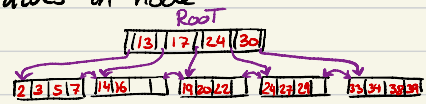
↑ point queries only

B+ trees

most common indexing type → Binary search tree

Dynamic, Balanced

4 values in node



Insertion to full node: split and move middle key to parent lvl

Clustered

Data is stored according to the index...

need to fetch fewer blocks from memory

Only 1 clustered index per table

Un-Clustered

Should only be used for VERY selective queries

Covering

Contains all the information for the query...

i.e. count(*) or select <column> where <column> is part of index

Multi-index

Firstly sorted by <attr1>, and secondly by <attr2>

good for point on <attr1> combined with range on <attr2>

Query Processing

Indexes

if covering: use

if clustered: use

if non-clustered: use ONLY when high selectivity

Multi-condition

use 2 non-clustered indices, return intersect (and) / union (or)

if 1 condition only returns few rows, scan 1 index, filter the "few"

use composite (& covering) indexing, return ü

if remaining columns are few (Because takes up memory)

Join

- evaluation only 1 & 2 in MySQL

Approach 1: Nested loop — for tuple in R, for tuple in S, if r.ID = s.ID, output(r,s) — $O(|R| \cdot |S|)$ — 2 × full table scans

Approach 2: Nested loop on index — easier to search index — binary $O(|R| \cdot \log|S|)$

Approach 3: Merge Join; sort both on join column, then merge the results!

works well for clustered & covering index

$O(|R| \log(|R|) + |S| \log(|S|) + |R| + |S|)$

sorting | merging

Approach 4: Hash Join; create hash table for 1 table, then scan through using hash function on other table to join — $O(|R| + |S|)$

works well when = fits in RAM

Apply non-join conditions before hashing

join cost + RAM for $\min(|R|, |S|)$

RDBMS Implementation

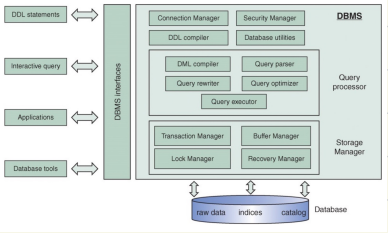
Relational DataBase Management System Implementation

Edgar F. CODD — Invented Relational Model while at IBM → made readable one-liners possible
1981 Turing Award

UPDATE <table> SET <column>=... WHERE <condition>

Jim Gray — Researched Transactions & Locking
1998 Turing Award Father of Transaction Processing

DBMS Architecture



Transactions ACID

- **A**tomicity; T is completed OR no effect at all
- **C**onsistency; T does not break integrity constraints
- **I**solation; T_1 & T_2 executed in parallel has same effect as sequential
- **D**urability; effect of committed T remains even if computer crashes

Implementing Ts

- C ~= satisfying constraints — Use indexes on PK & FK
- I = prevent corrupting changes — A1: Locking A2: Time Stamp Detection
- A & D = tracking changes — Logging 'Before' AND 'After' ☺

Logging (A&D → WAL) — Write-Ahead Logging

- ↳ Before doing changes we Force log record
- ↳ Before committing we Force I log record

Locking (I)

a Lock has a chosen modularity

Two-Phase Locking (2PL)

- ↳ Reading requires Shared S lock
- ↳ Writing requires write X lock

X locks cannot happen in conjunction with any other lock
All locks are released on commit/abort

Phantoms — a record sufficing conditions of X-locked records is a phantom
can be avoided using index locking to not allow insertion of phantom

Isolation Levels

1. Read uncommited: T can read uncommitted
2. Read commited: T can read committed, but might become overwritten later
3. Repeatable read: Phantoms are possible...
4. Serializable: scheduled, no anomalies!

main memory ☺

MMDBS, NoSQL & Scaling Out

OLTP vs. OLAP

only TRANSACTION processing vs. only ANALYTICS processing

* short, simple Ts
* Many users!

* complex queries
* few HEAVY users

Optimized Storage

Row-wise! Column-wise!

Here we can use MMDBM if all data fits in RAM...
less I/O waiting time

MMDBMS for OLTP

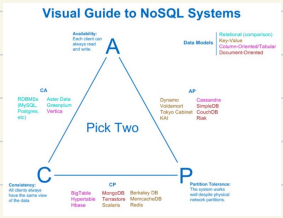
All data is kept in Main Memory... No waiting for disk I/O
Don't have to optimize for disk, BUT better cache utilization
We need to reduce log size on disk...

Scaling Up

to keep performance running decently, we distribute users on more web servers
↳ a single server has an upper bound on performance

NoSQL

Usually
↳ Not Relational, Schema-less structure
↳ Not only SQL (often SQL-like)
↳ Distributed Architecture*, ACID compliant**
↳ Mostly Open Source



Key-Value Stores like a dict ☺ f.ex. URL → HTML
Document Stores like JSON ☺
Graph Stores like nodes, edges & properties ☺

* Distributed → workload sharing, failure handling, balance (consistent hashing - virtual server concept)
** CAP Theorem → True, but is about trade-off when you have a partitioned network!

Big Data Management

Big Data

Data Science & Real Life Application

How are big amounts of data used/saved?

Particle Accelerator, DNA, Video Surveillance and so on...

Big Data is a collection of processes

↳ DBs, Docs, feeds, Analog

- collect, clean, ETL

The Three V's (Only Big Data if all are present)

↳ Volume! (amount)

↳ Velocity! (processing speed, accumulation speed) — data generation + processing action!

↳ Variety! (different types and sources!) — semi-structured!

+ Veracity! (how well we trust the data) — know WHOLE data! not just parts of it...

+ Value! (data provides monetary/societal value!)

Clustering Big Data

Identify groups in data (f.ex. k-means)

Classifying Big Data

Find Rules to predict outcomes (f.ex. Decision Trees)

Access Patterns

Distributed Storage + Large Sequential Scans

Run complex processing Pipelines!

RDBMS is not relevant to big data...

↳ no support for unstructured

↳ little support for distributed processing

Same ~for NoSQL options

MapReduce

1. map split input, 2. shuffle, 3. reduce, 4. gather

Example: Apache Hadoop (But! complex mapping, lots of disk writes)

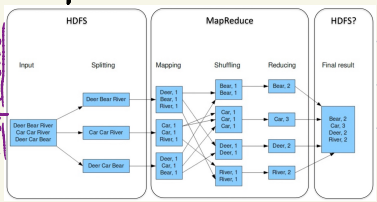
Spark

+Spark SQL

Transform one 'Resilient Distributed Dataset' (RDD) to another via operators...

Deep Pipelines + worker's memory sharing

MapReduce



Spark

- .map = one-to-one transformation
- .flatmap = one-to-any transformation
- .groupByKey = Hadoop's Shuffle
- .reduceByKey = Hadoop's Reduce
- .collectAsMap = collect to Master
- ...