

# Fun with databases

or: How I learned to stop worrying and love the deadlock

---

Mario.Lassnig@cern.ch

European Organization for Nuclear Research (CERN)

*<https://github.com/mlassnig/Databases-Introductory-Course>*

# About me

---

## ↪ Software engineer

- ↪ Data management for the ATLAS Experiment, CERN, since 2006
- ↪ Automotive navigation, AIT Vienna, 2004-2006
- ↪ Avionics for autonomous robots, Austrian Space Forum, since 2008

## ↪ Education in computer science

- ↪ Graph theory (Master's), Multivariate statistics and machine learning (PhD)

## ↪ Largest database built yet for the Rucio data management system

- ↪ 3.5+ billion rows
- ↪ 50+ tables, 4 TB
- ↪ 400 Hz / 35'000 IOPS

# About this course

---

- ↪ For every topic
  - ↪ We will do some theory
  - ↪ We will do some hands-on exercises
- ↪ The contents from the session codes are complete and will give you the solution.
  - ↪ Don't get tempted to copy/paste :-D
  - ↪ There'll be a challenge/exercise at the end, where you have to apply what you have learned.

Please interrupt me!

# Part I — Introduction

# CAP Theorem

---

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees [Brewer, 2000]

All clients always see the same data.  
**Consistency**

File systems, single-instance databases, ...

All clients can always read and write.  
**Availability**

Choose Two

Distributed databases

Web caching, DNS, ...

**Partition tolerance**  
The data can be split across the system.

# ACID and BASE

---

## ↪ ACID

- ↪ Atomicity      All or nothing operations
- ↪ Consistency      Always a valid state
- ↪ Isolation      Operations can be serialised
- ↪ Durability      Data is never lost

## ↪ BASE

- ↪ Basically available      More often than not
- ↪ Soft state      Data might be lost, but you will only know when you try to access it
- ↪ Eventually consistent      Your query might return old data

# So what is this NoSQL thing anyway?

---

- ↪ Term invented for a relational database without a SQL interface (Carlo Strozzi, 1998)
- ↪ Term re-coined by *last.fm* in 2009
  - ↪ At an open-source distributed databases workshop
  - ↪ What they actually meant: how to deal with the exponential increase in storage requirements
- ↪ Promises of NoSQL databases?
  - ↪ Relational model might not map well to application's data structures
  - ↪ Improve productivity by using non-relational stores instead as application backend
- ↪ Improve performance for "web-scale" applications?
  - ↪ Remember the CAP theorem
  - ↪ There is no free lunch

# Types of databases

---

## ↪ Row stores

↪ Oracle, PostgreSQL, MySQL, SQLite, ...

## ↪ Column stores

↪ Hbase, Cassandra, Hypertable, Monet, ...

## ↪ Data structure stores

↪ Elasticsearch, MongoDB, Redis,  
PostgreSQL, ...

## ↪ Many have overlapping concepts

## ↪ Get confused here:

<http://nosql-database.org/>

## ↪ Key/Value stores

↪ Dynamo, Riak, LevelDB, Kyoto, ...

## ↪ Graph stores

↪ Neo4j, Titan, Hypergraph, ...

## ↪ Multimodel stores

↪ ArangoDB, CortexDB, ...

## ↪ Object stores

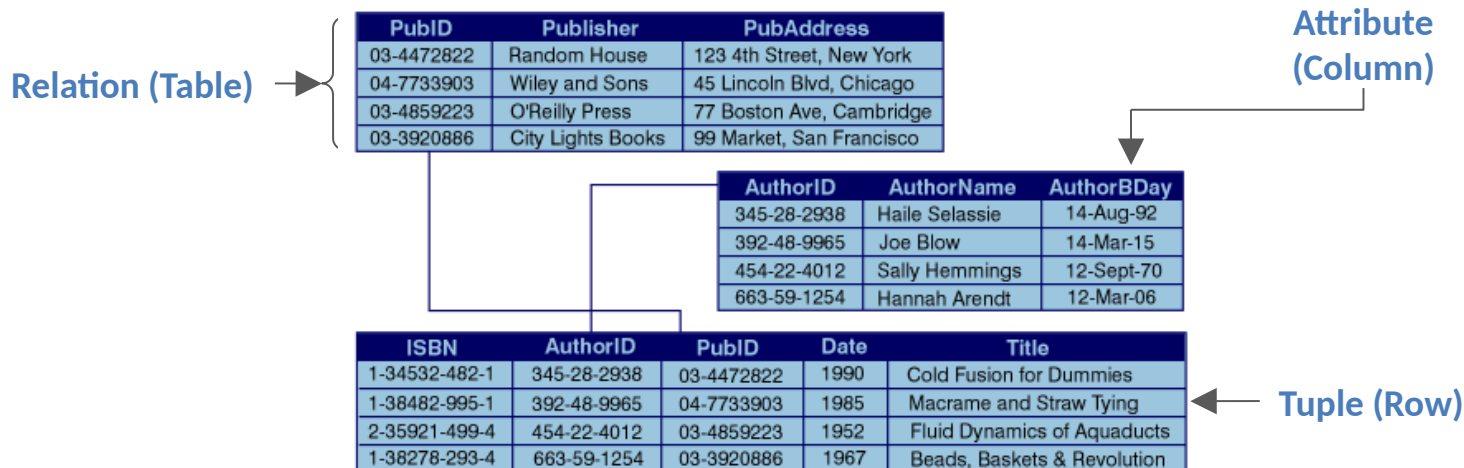
↪ Versant, Ceph, ...

|                              |
|------------------------------|
| Relational<br>Non-relational |
|------------------------------|



# Relational model (Edgar F. Codd, 1969)

|            |           |        |            |
|------------|-----------|--------|------------|
| ↪ Concept  | Relations | Tuples | Attributes |
| ↪ Database | Table     | Row    | Column     |



<http://www.ibm.com/developerworks/library/x-matters8/relat.gif>

# Structured Query Language (SQL)

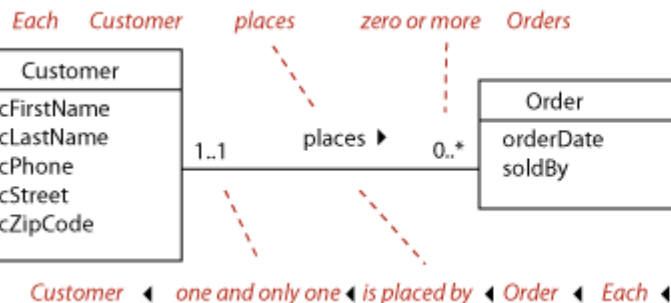
---

- ↪ Also proposed by Edgar F. Codd, 1969
- ↪ Interaction with DBMS (Database Management System) using declarative language
- ↪ Developed from SEQUEL language an ANSI/ISO Standard in 1986
- ↪ Ess Que Ell? Sequel?
  - ↪ Oracle defines pronunciation as "sequel", then we have "My-S-Q-L", and PostgreSQL is "Postgres"
  - ↪ Don Chamberlin, one of the original authors of the SEQUEL language suggests to go with ISO: S-Q-L

```
CREATE TABLE table_name;  
SELECT column_name FROM table_name WHERE column_name = value;  
INSERT INTO table_name(column_name) VALUES (value);  
UPDATE table_name SET column_name = value;  
DELETE FROM table_name WHERE column_name = value;  
DROP TABLE table_name;
```

# Row Stores

- ↳ Your classic RDBMS
- ↳ Physically stores data row-by-row
- ↳ Easy joining of data between tables
  - ↳ one-to-one
  - ↳ one-to-many
  - ↳ many-to-many
- ↳ Normalization algorithms to reduce duplicate data and complexity
- ↳ Not so good for aggregation (RDBMS vendors compete here)



| Orders |             |
|--------|-------------|
| custID | orderDate   |
| 5678   | 14-JUL-2003 |
| 9012   | 14-JUL-2003 |
| 5678   | 18-JUL-2003 |
| 5678   | 20-JUL-2003 |

copied values

OrderLines

| custID | orderDate   | UPC         |
|--------|-------------|-------------|
| 5678   | 14-JUL-2003 | 51820 33622 |
| 9012   | 14-JUL-2003 | 51820 33622 |
| 9012   | 14-JUL-2003 | 11373 24793 |
| 5678   | 18-JUL-2003 | 81809 73555 |
| 5678   | 20-JUL-2003 | 51820 33622 |
| 5678   | 20-JUL-2003 | 81809 73555 |
| 5678   | 20-JUL-2003 | 81810 63591 |

copied values

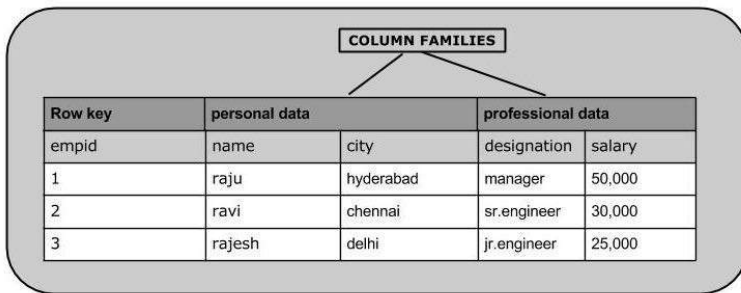
Products

| UPC         |
|-------------|
| 51820 33622 |
| 42877 34040 |
| 81809 73555 |
| 11373 24793 |
| 81810 63591 |

<http://www.tomjewett.com/dbdesign/dbdesign.php?page=manymany.php>

# Column Stores

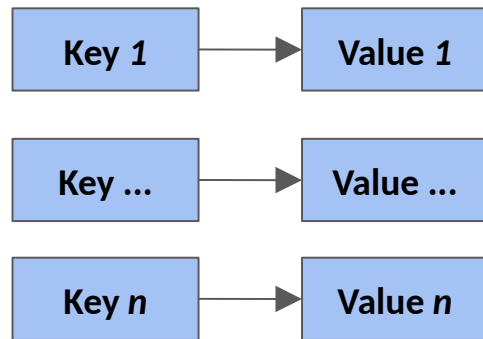
- ↪ Many applications do not need relations
- ↪ Row-based systems like traditional RDBMS are ill-suited for aggregation queries
  - ↪ Things like SUM/AVG of a column?
  - ↪ Needs to read blocks of columns, sometimes full rows, unnecessarily
- ↪ Physical layout of data column-wise instead
  - ↪ saves IO and improves compression, facilitates parallel IO
  - ↪ makes joins between columns harder
- ↪ Need to save on joins somehow
  - ↪ Group columns which get queried together
  - ↪ Column families



# Key/Value Stores

---

- ↳ Hashmap for efficient insert and retrieval of data
  - ↳ You might know this as an associative array, or dictionary, or hashtable, ...
- ↳ Keys and values are usually bytestreams, but practically just strings
- ↳ Usually there are some performance guarantees, via options like
  - ↳ sorted keys
  - ↳ length restrictions
  - ↳ different hash functions
- ↳ Simple and easy to use
  - ↳ Either as compile-time libraries
  - ↳ Or as a server, usually via wrapped native protocols, REST, ...
- ↳ First Key/Value store: dbm, 1979



# Data Structure Stores

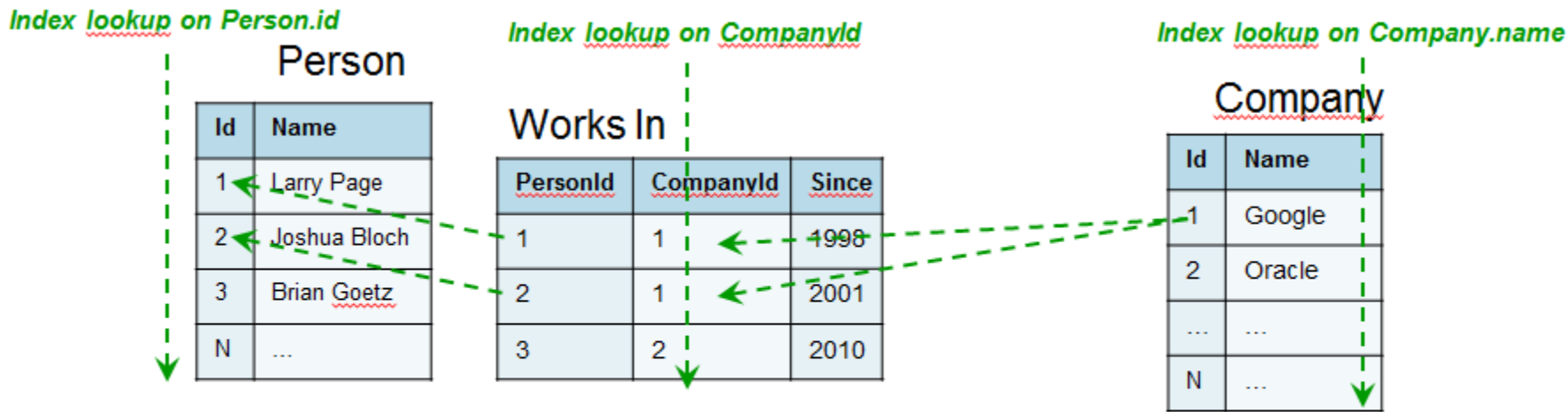
- ↪ Basically key/value stores, with the added twist that the stores knows something about the internal structure of the value
- ↪ Very easy to use as a backend for an application
- ↪ When people think NoSQL, this is usually what they mean



<http://docs.mongodb.org/v3.0/images/data-model-denormalized.png>

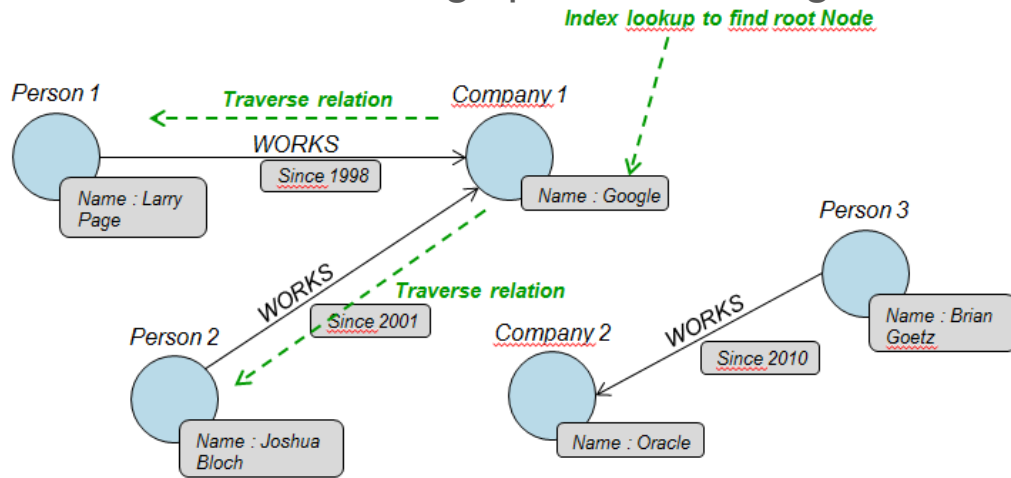
# Graph Stores

↪ In the relational world, actual *n-to-n* relations are cumbersome to model (despite the name...)



# Graph Stores

- ↪ Make relations first-class citizens
- ↪ Physical layout optimised for distance between data points
- ↪ Leads to easy & fast traversal for the graph database engine





# Hands-on Session 1

---

↪ Create, read, update, delete data

↪ Using C/C++ and Python

↪ On

|                        |                |                |
|------------------------|----------------|----------------|
| ↪ <b>PostgreSQL</b>    | relational     | row-based      |
| ↪ <b>LevelDB</b>       | non-relational | key/value      |
| ↪ <b>Redis</b>         | non-relational | data structure |
| ↪ <b>ElasticSearch</b> | non-relational | data structure |
| ↪ <b>Neo4j</b>         | non-relational | graph          |

## Part II — Fun and profit

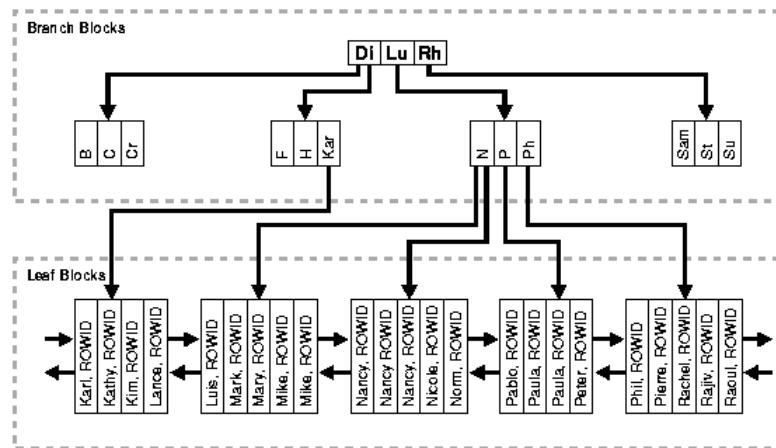
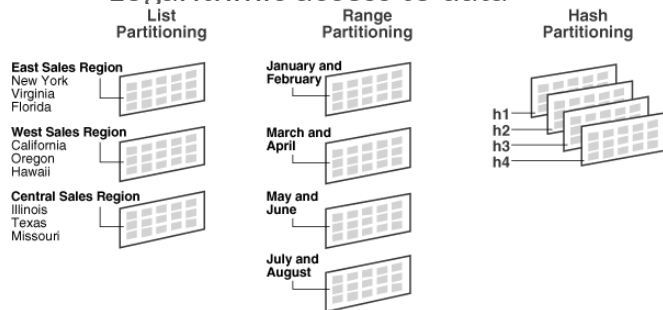
# Query plans

---

- ↪ IMHO the single most important thing when dealing with databases
- ↪ You want to avoid going to disk, to reduce the IOPS and CPU usage
- ↪ In order of excessiveness
  - ↪ FULL TABLE SCAN
  - ↪ PARTITION SCAN
  - ↪ INDEX RANGE SCAN
  - ↪ PARTITION INDEX RANGE SCAN
  - ↪ INDEX UNIQUE SCAN
  - ↪ PARTITION INDEX UNIQUE SCAN
- ↪ Not all FULL TABLE SCANS are bad
  - ↪ If you need to retrieve a lot of data, and it is indexed, you will get random IO on the disk. Much better to prefer serial scan (FULL, PARTITION) in such cases
  - ↪ If your data is low cardinality (few values, lots of rows) then indexes won't help and slow down INSERTs

# Query plans — How to optimise?

- ↳ Understand the EXPLAIN PLAN statement, then decide
- ↳ Partitions
  - ↳ Physical separation of data
  - ↳ Costly to introduce afterwards (usually requires downtime with schema migration)
- ↳ Indexes
  - ↳ Either global or partition local
  - ↳ Logarithmic access to data



# Transactional safety

↪ In multi-threaded environments concurrent access to the same data is likely

↪ Dirty Read

↪ Read data by uncommitted transaction

↪ Non-Repeatable Read

↪ Reads previously read data again, but it has changed in the meantime by another transaction

↪ Phantom Read

↪ Repeated query of the same conditions yields different results due to intermediate other transaction

| Isolation Level  | Dirty Read   | Nonrepeatable Read | Phantom Read |
|------------------|--------------|--------------------|--------------|
| Read uncommitted | Possible     | Possible           | Possible     |
| Read committed   | Not possible | Possible           | Possible     |
| Repeatable read  | Not possible | Not possible       | Possible     |
| Serializable     | Not possible | Not possible       | Not possible |

↪ Different transaction isolation levels provide safeguards

↪ By locking of rows, or sometimes even tables, and thus making other transactions wait

↪ The more you lock, the slower you are

↪ Can lead to deadlocks — always lock rows in the same order!

# Sparse metadata

---

↪ Think "tagging"/"labelling" datasets — Difficult problem in relational model

↪ Keep extra columns or implement a relational key/value store?

↪ Extra columns are bad for physical disk layout

↪ Relational key/value store requires lots of joins — not good for cache and CPU

↪ Will you ever search on metadata?

↪ **No** Store the metadata as a JSON-encoded string in a single column

↪ **Yes** Many different kinds of metadata?

→ **No** Maintain a separate metadata table, with pre-created columns

→ **Yes** Use the built-in JSON support of Oracle or PostgreSQL  
or alternatively use a non-relational database and keep the data in sync

↪ There is some really bad advice on StackOverflow promoting a generic key/value model for relational databases — Caveat emptor!

# Competitive locking

---

- ↪ Many applications have the following use case
  - ↪ Many processes write something in a queue — the "backlog" of things to do
  - ↪ Many processes read from that queue — and work it off in parallel
- ↪ Scheduling problem
  - ↪ Do things in order? Prioritise certain things?
  - ↪ How to avoid that multiple workers process the same things?
  - ↪ Repeat after me: a database is not a queuing system
- ↪ However, sometimes a DB is all you have... And there are two potential solutions
- ↪ Database-level (row read lock, easy):
  - ↪ BEGIN;      SELECT row FOR UPDATE;      COMMIT/ROLLBACK;
- ↪ Application-level (no lock, complex)
  - ↪ When selecting work, compute the row-hash, convert to integer, take modulo #workers
  - ↪ Then only process rows that match the worker-id

## Part III — Survival



# Distributed transactions

- ↪ Sometimes you really need two different data stores
- ↪ Sometimes you need to be consistent between both
- ↪ Consensus protocols are needed
  - ↪ Two-phase commit
  - ↪ Paxos
- ↪ Needs operational support by DBMS
  - ↪ pending writes / write-ahead log (WAL)
- ↪ But you still have to code it in the application
  - ↪ This is where you will lose performance

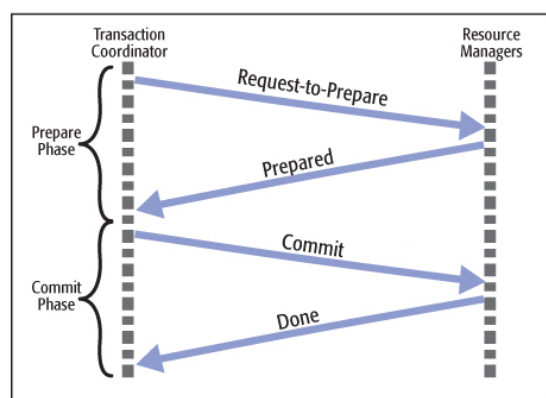
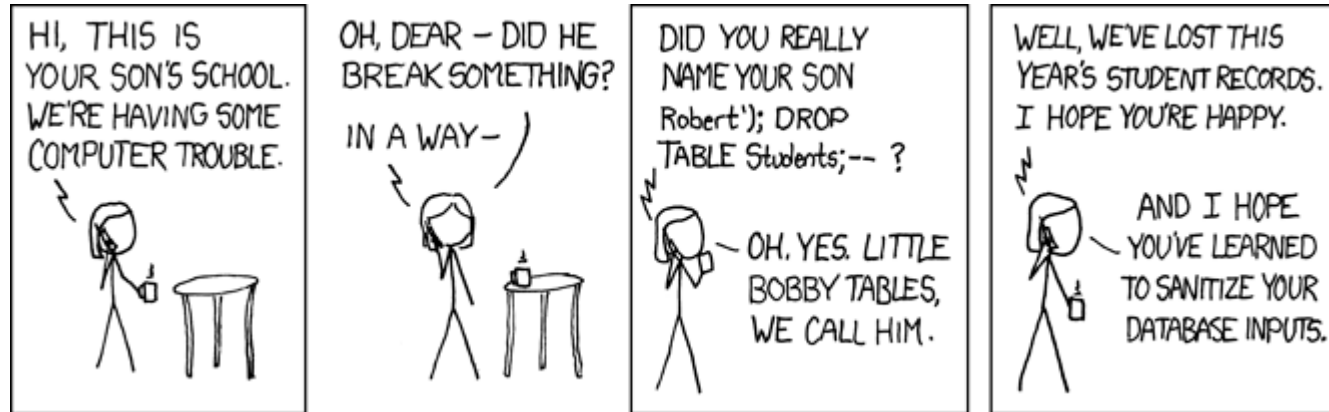


Figure 1 • The two-phase commit protocol

# SQL Injection



# SQL Injection

---



<https://hackaday.com/files.wordpress.com/2014/04/18mpenleoksg8jpg.jpg?w=636>

# Application level woes

---

- ↪ Handling sessions — this will be your major source of pain
  - ↪ Connections, Sessions, Transactions (and how they are differently named/treated by each DBMS)
- ↪ Most databases only have a limited number of available connections, or sessions
- ↪ Some pay with CPU for logons, others with RAM for sessions, etc...
- ↪ E.g., if you have to channel 100 concurrent transactions across 10 connections
  - ↪ SessionPooling/QueuePooling — every language/database has its own idea
  - ↪ Takeaway message: Use an abstraction, and don't code it yourself
- ↪ Python                  SQLAlchemy, Django
  - ↪ Also comes with Object-Relational Mapper
  - ↪ Automatically translates Python objects into the relational model
- ↪ C++                        CodeSynthesisODB
  - ↪ Also supports BOOST datatypes!

## Part IV — The Challenge

# Challenge description

---

- ↪ Write a Twitter clone before time runs out
- ↪ Choose any database you like, the VMs are preconfigured for all of them
- ↪ Stick to the following UX — you will need to write four programs
  - ↪ `Inserter`                      periodically inserts new random tweet into the database
  - ↪ `Latest`                        periodically print the latest 10 tweets
  - ↪ `Random`                        periodically print random 5 tweets
  - ↪ `Stats`                         periodically print statistics about the database content
    - ↪ How many tweets were added in the last minute by each user and overall (insertion rate)
    - ↪ How often a given tweet was displayed (popularity of a particular tweet)
  - ↪ Start 10 `inserter`, 10 `latest`, 10 `random`, 1 `stat`
- ↪ Use this random tweet/sentence generator (pip install loremipsum)
  - ↪ `import loremipsum                      loremipsum.generate_sentence()`
- ↪ When stuck, ask me — when done, show me! Good luck and have fun!