Databases

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

Mario.Lassnig@cern.ch
European Organization for Nuclear Research (CERN)

About me

- Software Engineer
  - Data Management & Analytics for the ATLAS Experiment, CERN, 2006-ongoing
  - Automotive Navigation, AIT Vienna, 2004-2006
  - Avionics for autonomous robots, Austrian Space Forum, 2008-ongoing
- Education
  - Multivariate statistics and machine learning (PhD)
  - Graph theory (Master's)
  - Cryptography (Undergrad)
- Largest 24/7 database built yet
  - 3+ billion rows
  - 35'000 IOPS

About this course

- For every topic
  - We will do some theory
  - We will do some hands-on exercises

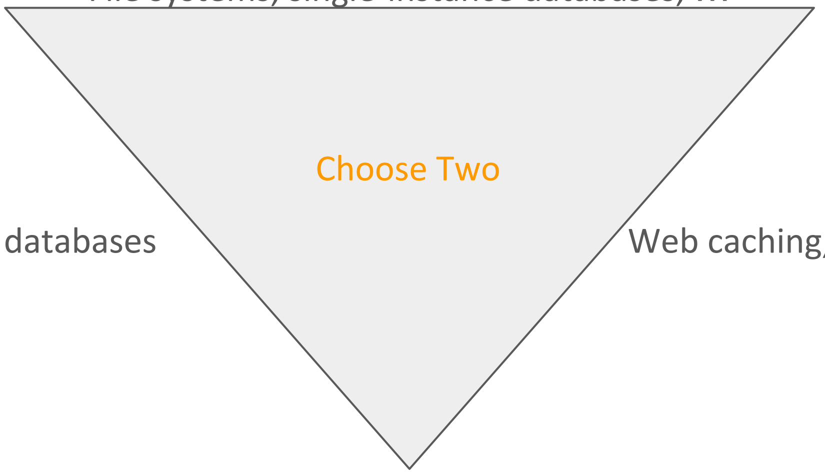
- The contents from the session codes are complete and will give you the solution.
  - Don't blindly copy/paste, try to understand what's going on.
  - There'll be a challenge/exercise at the end, where you have to apply what you have learned.

Please interrupt me whenever necessary!

Part I — Introduction

CAP Theorem

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



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?

- Carlo Strozzi, 1998
- Term invented for a relational database without a SQL interface

- 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, …
- Key/Value stores
  - Dynamo, Riak, LevelDB, Kyoto, …
- Graph stores
  - Neo4j, Titan, Hypergraph, …
- Multimodel stores
  - ArangoDB, CortexDB, …
- Object stores
  - Versant, Ceph, …

- Many have overlapping concepts
- Get confused here: http://nosql-database.org/

Relational
Non-relational

Relational model

- Proposed by Edgar F. Codd, 1969
- **Concept** Relations Tuples Attributes
- **Database** Table Row Column

Attribute/Column

Relation/Table

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sept-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1952	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution

Tuple/Row

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

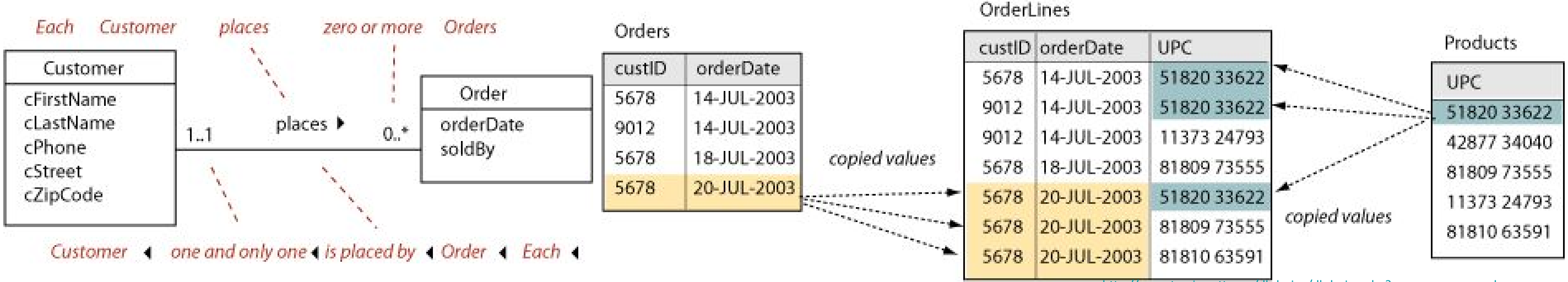
Structured Query Language (SQL)

- 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

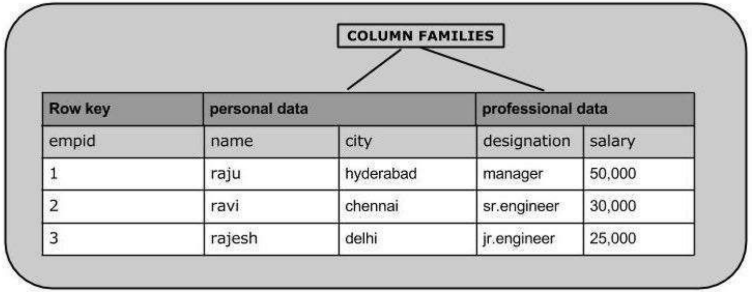
- Your classic RDBMS
- Physically stores data row-by-row
- Easy joining of data between tables
  - one-to-one
  - one-to-many
  - many-to-many
- Normalization algorithms to reduce duplicate data and complexity
- Not so good for aggregation (RDBMS vendors compete here)



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
  - native support for column-families

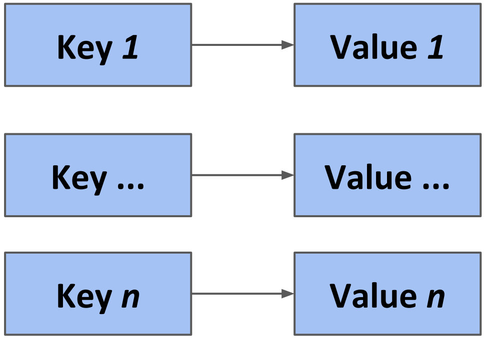


Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

http://www.tutorialspoint.com/hbase/images/table.jpg

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, or via REST
- First Key/Value store: dbm, 1979



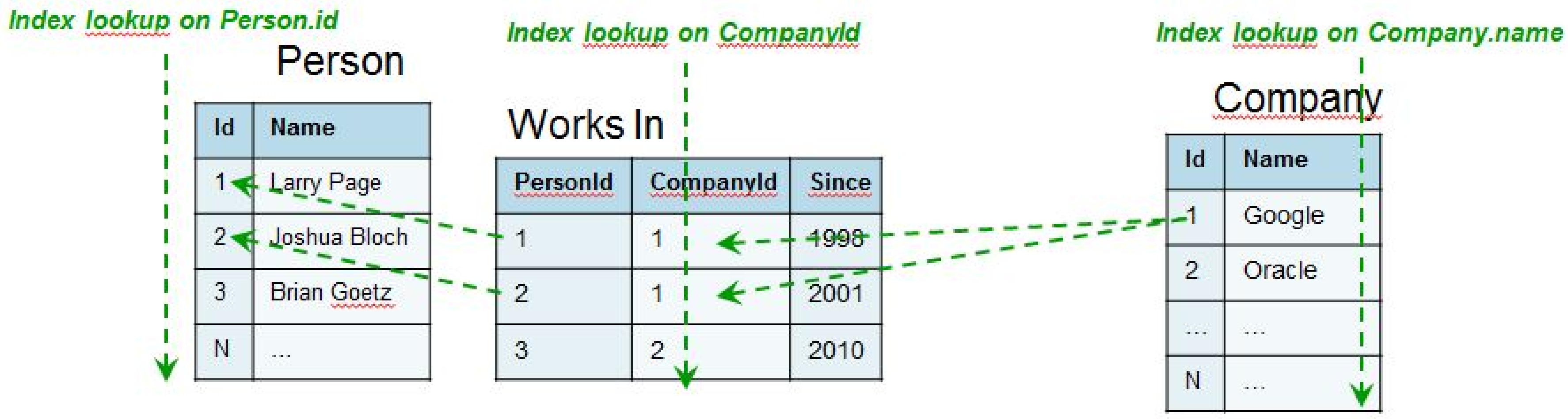
Data Structure Stores

- 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

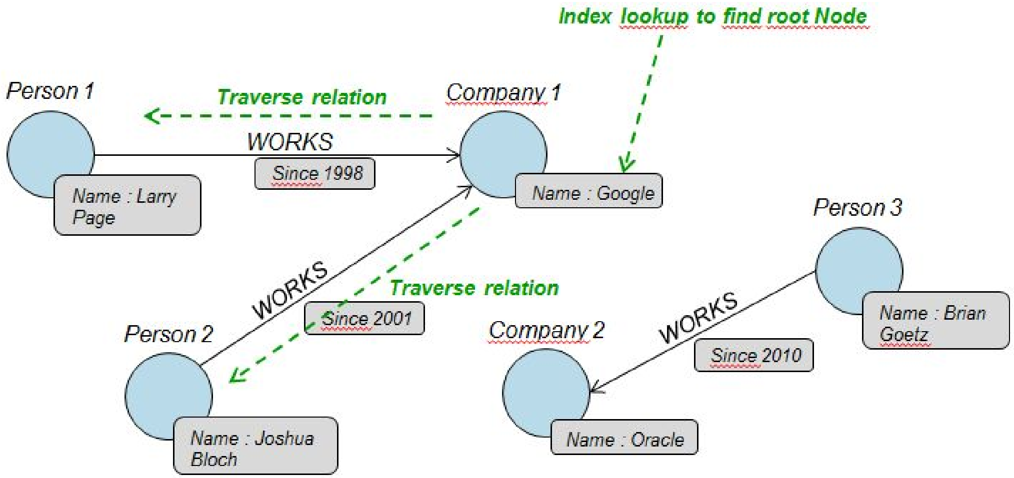
↪ In the relational world, actual *n-to-n* relations are cumbersome to model (despite the name…)



http://blog.octo.com/wp-content/uploads/2012/07/RequestInSQL.png

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



http://blog.octo.com/wp-content/uploads/2012/07/RequestInGraph.png

Hands-on Session 1

- Create, read, update, delete data
- Using C/C++ and Python
- On
  - **PostgreSQL** relational row-based
  - **LevelDB** non-relational key/value
  - **Redis** non-relational data structure
  - **ElasticSearch** non-relational data structure
  - **Neo4j** non-relational graph

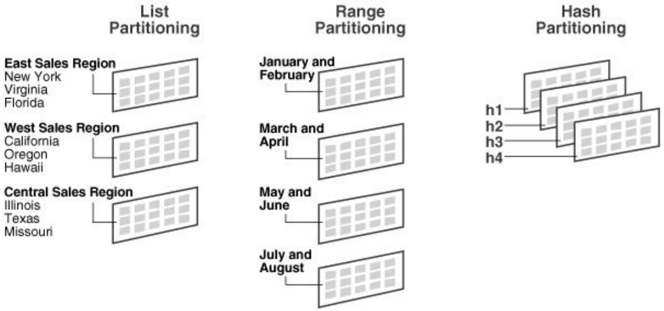
Part II — Fun and profit

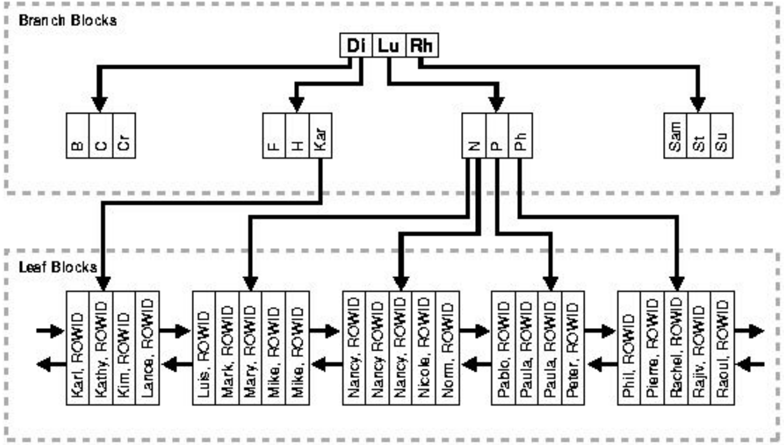
Query plans

- 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 you 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





https://docs.oracle.com/cd/B19306_01/server.102/b14220/img/cncpt158.gif

http://www.mattfleming.com/files/images/example.gif

Transactional safety

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

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

- 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 some conditions yields different results due to intermediate other transaction

- 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 if the developer is careless — 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

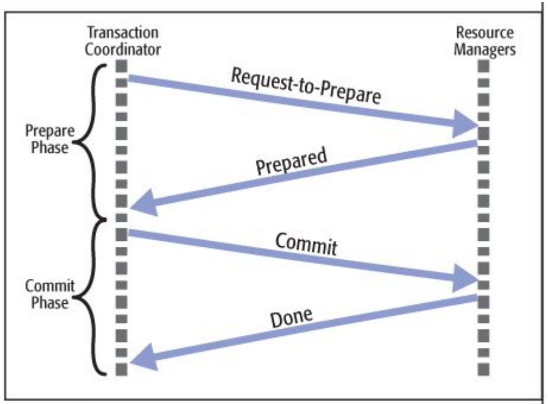
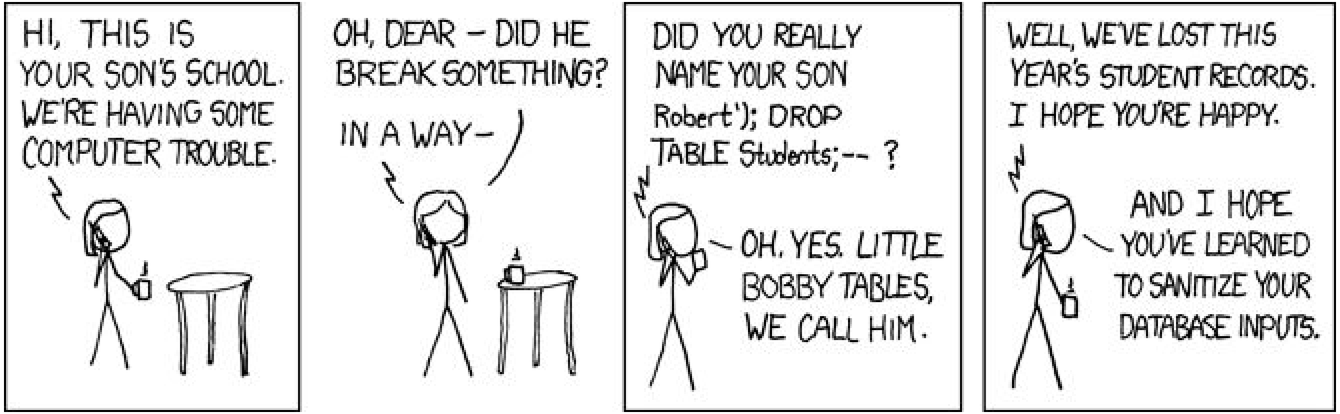


Figure 1 • The two-phase commit protocol

http://gemsres.com/photos/story/res/43755/fig1.jpg

SQL Injection



https://xkcd.com/327/

SQL Injection

https://hackadaycom.files.wordpress.com/2014/04/18mpenleoksq8jpg.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 y ourself
- 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 twwet)
  - 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!

Databases

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

Mario.Lassnig@cern.ch

European Organization for Nuclear Research (CERN)

CERN