

Application development with relational and non-relational databases

Mario Lassnig

European Organization for Nuclear Research (CERN)

mario.lassnig@cern.ch

About me

- Software Engineer
 - Data Management & Analytics for the ATLAS, CERN, 2006-ongoing
 - Automotive navigation, AIT Vienna, 2004-2006
 - Avionics for autonomous robots, Austrian Space Forum, 2008-ongoing
- Education
 - Cryptography (Undergrad)
 - Graph theory (Master's)
 - Multivariate statistics and machine learning (PhD)
- Largest 24/7 database built yet
 - 3+ billion rows
 - 30'000 IOPS

About this course

- For every topic
 1. We will do some theory
 2. We will do a hands-on session
- Please don't blindly copy and paste the session codes from the wiki during the hands-on sessions; there'll be exercises later where you'll have to use what you've learned!

Please interrupt me whenever necessary!

Part I – Introduction

- Relational primer
- Non-relational primer
- Data models

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

All clients can always read and write
Availability

File systems, single-instance databases, ...

Choose two.

Distributed databases

Web caching, DNS, ...

Partition tolerance

The data can be split across the system

ACID and BASE

ACID

- **A**tomicity
All or nothing operations
- **C**onsistency
Always valid state
- **I**solation
Operations can be serialised
- **D**urability
Data is never lost

BASE

- **B**asically **a**vailable
More often than not
- **S**oft state
Data might be lost
- **E**ventually consistent
Might return old data

So what is this NoSQL thing?

- Carlo Strozzi, 1998
- Term invented for a relational database without a SQL interface
- Term re-coined 2009 by last.fm
 - At an open-source distributed databases workshop
 - Deal with the exponential increase in storage requirements
- Improve programmer productivity
 - Relational model might not map well to application native data structures
 - Use 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

Relational
Non-relational

- Row Stores
 - Oracle, PostgreSQL, MySQL, SQLite, ...
- Column Stores
 - Hbase, Cassandra, Hypertable, MonetDB ...
- Document Stores / Data Structure Stores
 - ElasticSearch, MongoDB, CouchDB, Redis, PostgreSQL ...
- Key/Value Stores
 - Dynamo, Riak, LevelDB, BerkeleyDB, Kyoto, ...
- Graph Stores
 - Neo4j, Titan, Hypergraph, ...
- Multimodel Stores
 - ArangoDB, CortexDB, ...
- Object Stores
 - Versant, ...
- Many actually have overlapping concepts
- Get confused here: <http://nosql-database.org/>

Relational model

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

Relation

Hypothetical Relational Database Model

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

Attribute

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

Tuple

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

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

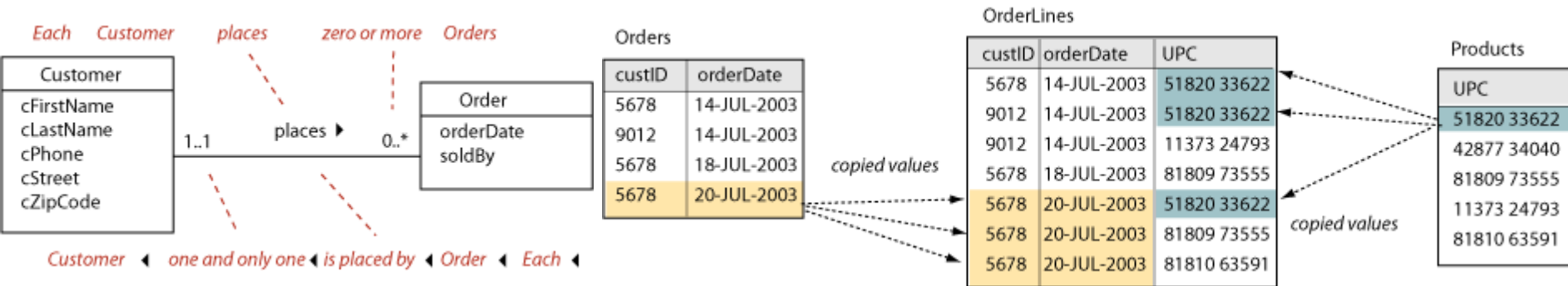
Structured Query Language

- Proposed by Edgar F Codd, 1970
- Interaction with DBMS using declarative programming language
- ANSI/ISO Standard since 1986
- Ess Que Ell? Sequel?

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

Row Stores

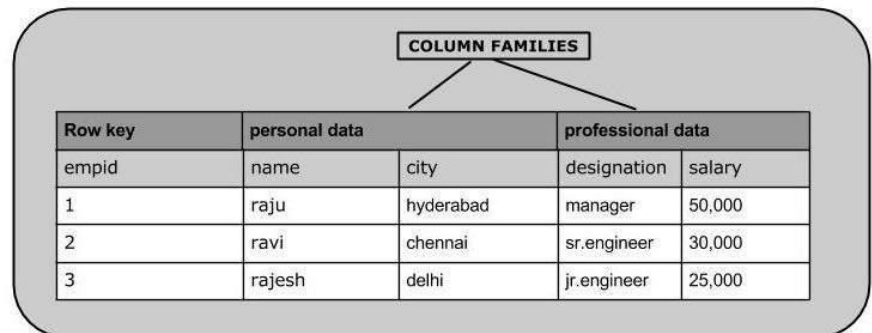
- Your classic RDBMS
- Physically stores data row-by-row
- Easy joining of data between tables
 - one-to-one
 - one-to-many
 - many-to-many
- Normalization procedures to reduce duplicate data and complexity
- Not so good for aggregation (RDBMS vendors compete here)



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

Column Stores (the most confusing of all)

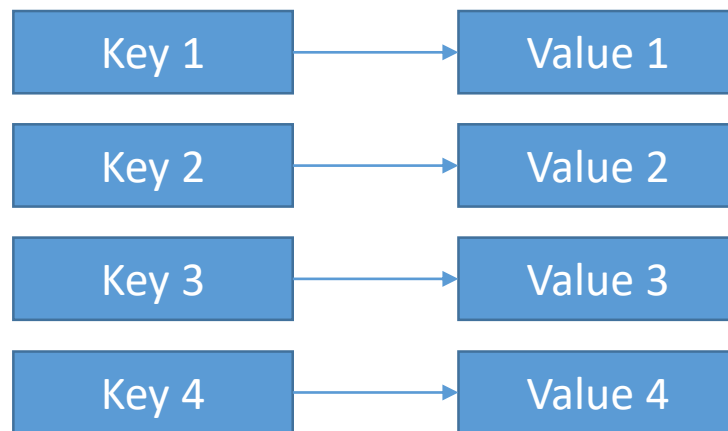
- Many applications do not need relations, think analytics...
- Row-based systems like traditional relational databases are ill-suited for aggregation queries
 - Things like SUM/AVG of a column?
 - Needs to read full row unnecessarily
- Physical layout of data column-wise instead
 - Saves IO and improves compression, facilitates parallel IO
 - Makes joins between columns harder
 - Organize columns in column-groups/families to save joins
 - Most column stores have native support for column-families



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

Key/Value Stores

- Hashmap for efficient insert and retrieval of data
 - You might know this as associative array, or dictionary, or hashtable
- Keys and value usually are bytestreams, but practically just strings
- Usually there are some performance guarantees, via options like
 - Sorted keys
 - Length restrictions
 - Hash functions
- Simple and easy to use
 - Either as compile-time library
 - Or as server, usually via wrapped native protocols, or via REST
- First one: dbm, 1979



Document Stores / Data Structure Stores

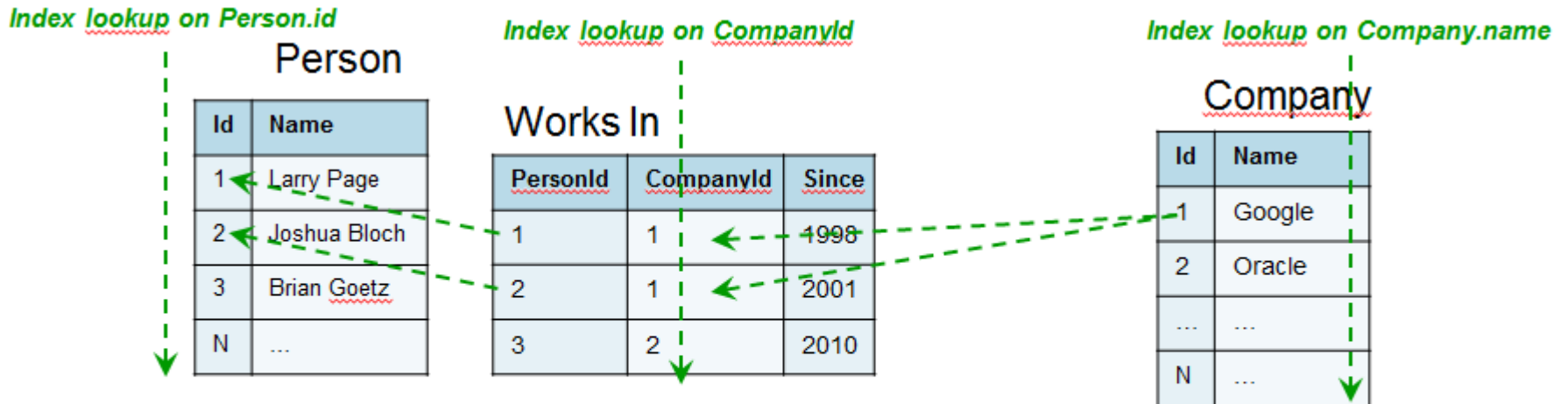
- Basically key/value stores, with the added twist that the store knows something about the internal structure of the value
- Very easy to use as backend for application
- When people think NoSQL, this is usually what they mean
- This flexibility comes at a price though – we'll discuss this later



http://docs.mongodb.org/v3.0/_images/data-model-denormalized.png

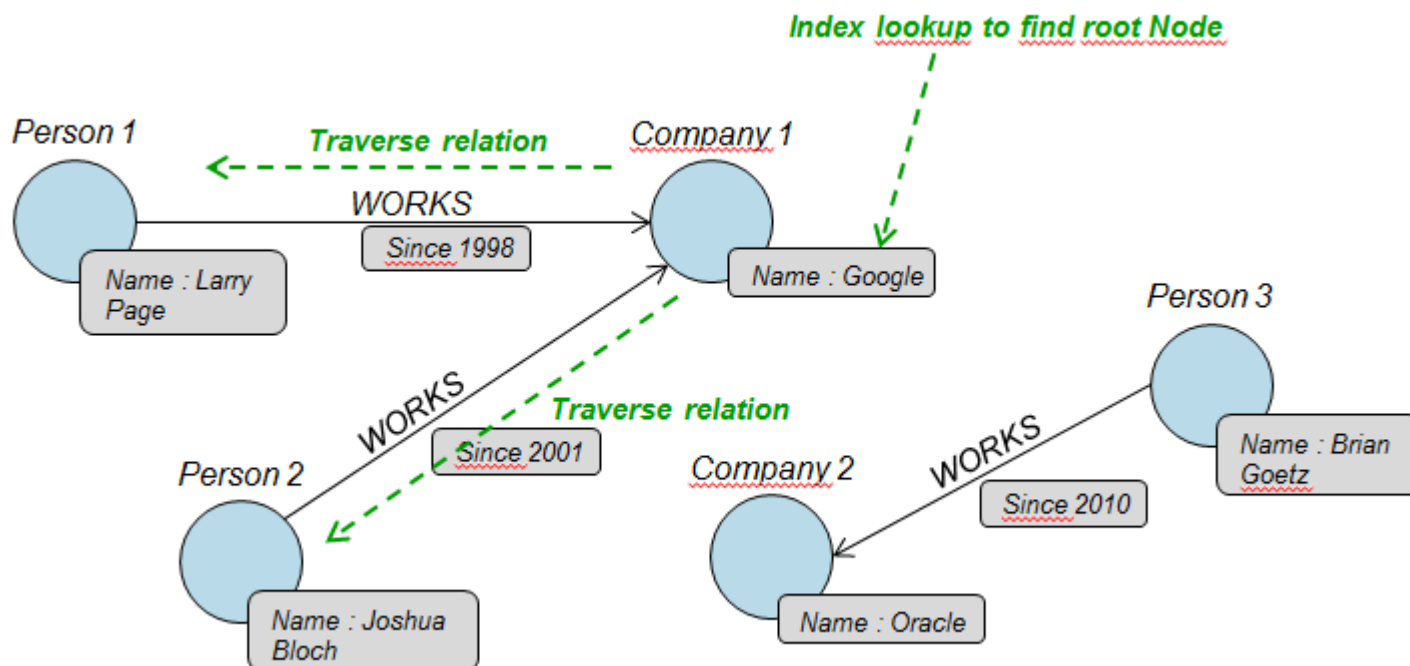
Graph Stores

- In the relational model actual n-to-n relations are cumbersome



Graph Stores

- Make relations first-class citizens
- Physical layout optimised for “distance” between data points
- Leads to easy & fast traversal for 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)
 - MonetDB (relational – column-based)
 - LevelDB (nonrelational – key/value)
 - Redis (nonrelational – data structure)
 - MongoDB (nonrelational – document)
 - ElasticSearch (nonrelational – document)
 - Neo4j (nonrelational – graph)

https://wiki.scc.kit.edu/gridkaschool/index.php/Relational_and_Non-relational_Databases

Part II – Fun and profit

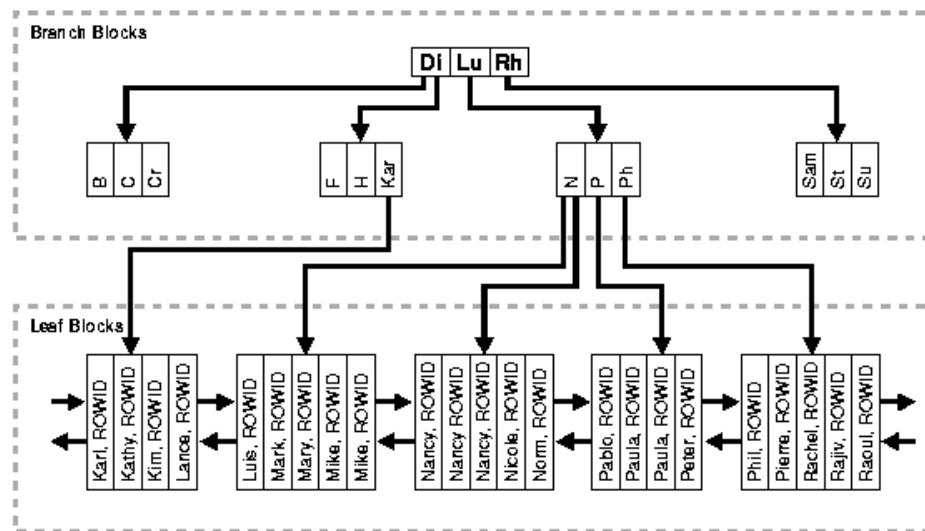
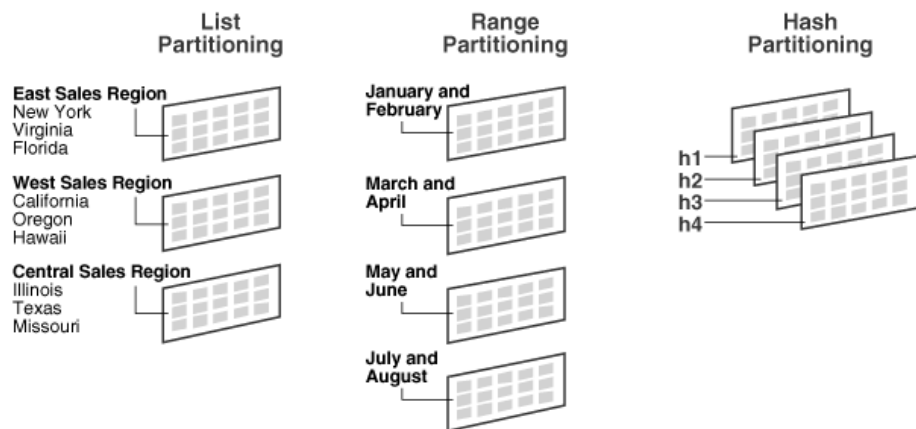
- Query plans and performance tuning
- Transactional safety in multi-threaded environments
- Sparse metadata
- Competitive locking and selection

Query plans

- The single most important thing you learn today
- You want to avoid going to disk, to reduce number of IOPS and CPU
- 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 – prefer serial scan (FULL, PARTITION) in such cases
 - If your data is of low cardinality (few values, lots of rows), then indexes will not help

Query plans – How to optimize?

- Understand EXPLAIN PLAN statement, then decide
- Partitions
 - Physical separation of data
 - Costly to introduce afterwards (usually requires schema migration)
- Indexes
 - Either global or partition local
 - Log-n 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

PostgreSQL prohibits this by design

- In multi-threaded environments concurrent access to the same data is likely – this can cause serious problems

- 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

- Different transaction isolation levels provide safeguards

- By locking of rows and thus making other transactions wait
 - The more you lock, the slower you are
 - Can lead to deadlocks if careless – always lock rows in the same order!



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

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 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 as a last resort: use a non-relational database
- There is some really bad advice on StackOverflow promoting a “generic” approach to metadata – please don’t do this

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 – process them 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
- There are two potential solutions – each with their own drawback
- Database-level (row read lock, easy):
`BEGIN; SELECT row FOR UPDATE ; COMMIT/ROLLBACK`
- Application-level (no lock, complex)
 - When selecting work, compute row-hash, convert, modulo #workers
 - Only work on rows that match worker-id

Part III – Survival

- Distributed transactions
- SQL injection
- Application-level woes

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 database (pending writes)
- But you still have to code it in the application

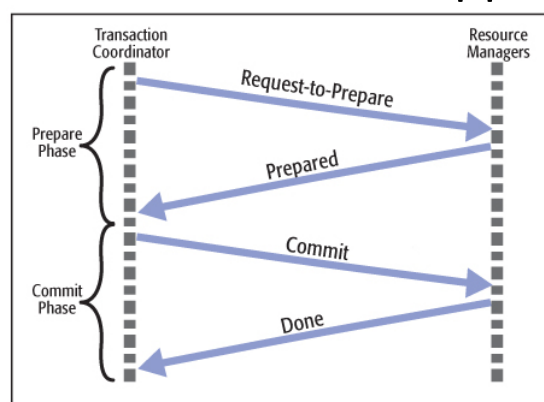


Figure 1 • The two-phase commit protocol

<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
 - Connection – Session – Transaction
- Most databases only have a limited number of available connections
- Some pay with CPU for logons, others with RAM for sessions, etc...
- E.g, have to channel 100 concurrent transactions across 10 connections
- SessionPooling/QueuePool – every language/database has it's own idea
- Use an abstraction, don't code this yourself

- SQLAlchemy, Django (Python)
 - Also come with an Object-Relational Mapper
 - Makes relations into transparent Python objects
- CodeSynthesisODB (C++)
 - Also supports BOOST datatypes(!)

Part IV – The challenge

Challenge description

- Write a Twitter clone before time runs out – 18:00
- Choose any database you like (after you thought about the design!)
- Stick to the following UX – you will write four programs
 - Inserter: periodically inserts new random tweet into the database
 - Latest: periodically prints the latest 10 tweets
 - Random: periodically prints random 5 tweets
 - Stats: periodically displays statistics
 - 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 stats
- Use this random sentence generator (`pip install loremipsum`)

```
import loremipsum
loremipsum.generate_sentence()
```
- When stuck, ask me – when done, show me! Good luck and have fun!

Application development with relational and non-relational databases

Mario Lassnig

European Organization for Nuclear Research (CERN)

mario.lassnig@cern.ch



www.cern.ch