Databases and Data Warehouses

Lecture BigData Analytics

Julian M. Kunkel

julian.kunkel@googlemail.com

University of Hamburg / German Climate Computing Center (DKRZ)

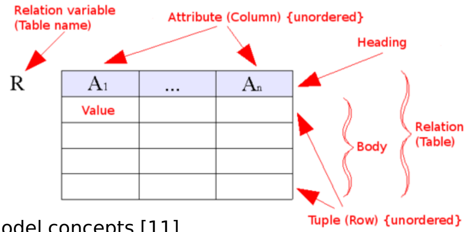
2017-11-03

Disclaimer: Big Data software is constantly updated, code samples may be outdated.

Outline

Relational Model [10]

- Database model based on first-order predicate logic
  - Theoretic foundations: relational algebra and calculus
- Data is represented as tuples
- Relation/Table: groups similar tuples
  - Table consists of rows and named columns (attributes)
  - No duplicates of complete rows allowed
- In a pure form, no support for collections in tuples
- Schema: specify structure of tables
  - Datatypes (domain of attributes)
  - Organization and optimizations
  - Consistency via constraints



Source: Relational model concepts [11]

Example Schema for our Students Data

Description

Database for information about students and lectures

Relational model

Matrikel	Name	Birthday
242	Hans	22.04.1955
245	Fritz	24.05.1995

Student table

ID	Name
1	Big Data Analytics
2	Hochleistungsrechnen

Lecture table

Matrikel	LectureID
242	1
242	2
245	2

Attends table representing a relation

Relationships

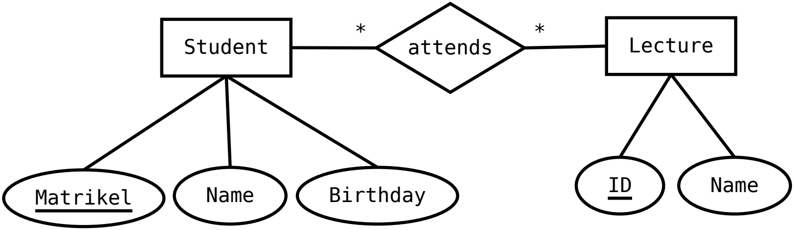
- Model relationships between data entities
- Cardinality defines how many entities are related
  - One-to-many: One entity of type A with many entities of type B
  - Many-to-many: One-to-many in both directions
  - One-to-one: One entity of type A with at most one entity of type B
- Relationships can be expressed with additional columns
  - Packing data of entities together in the table
  - Alternatively: provide a “reference” to other tables

Matrikel	Name	Birthday	Lecture ID	Lecture Name
242	Hans	22.04.1955	1	Big Data Analytics
242	Hans	22.04.1955	2	Hochleistungsrechnen
245	Fritz	24.05.1995	2	Hochleistungsrechnen

Student table with attended lecture information embedded

Entity Relationship Diagrams

- Illustrate the relational model and partly the database schema
- Elements: Entity, relation, attribute
  - Additional information about them, e.g., cardinality, data types



A student/lecture example in modified Chen notation
* is the cardinality and means any number is fine

Keys [16, 17, 18]

- A Superkey[1] allows addressing specific tuples in a table
- Superkey: Set of attributes that identify/address each tuple in a table
  - There can be **at most one tuple for each possible key value**
  - A superkey does not have to be minimal
    - e.g., all columns together are a Superkey of any table
    - After removing an attribute, it can still be a key
  - Simple key: key is only one attribute
  - Compound key: consists of at least two attributes
- Candidate key: a minimal key, i.e., no attribute can be removed
- **Primary key**: the selected candidate key for a table
- Foreign key: inherited key of another table
- Natural key: key that naturally is unique, e.g., matrikel
- Surrogate key: artificial key, e.g., numeric ID for a row

[1] Often it is just called key

Example Keys

Student table

Matrikel	Name	Birthday	...
242	Hans	22.04.1955	
245	Fritz	24.05.1995	

Lecture table

ID	Name	Semester
1	Big Data Analytics	SS15
2	Hochleistungsrechnen	WS1516

Attends table representing a relation

Matrikel	LectureID
242	1
242	2
245	2

- Student table
  - Candidate keys: Matrikel, (name, birthday, city), social insurance ID
  - Primary key: Matrikel (also a natural key)
- Lecture table
  - Candidate keys: ID, (Name, Semester)
  - Primary key: ID (also a Surrogate Key)
- Attends table
  - Candidate key: (Matrikel, Lecture ID)
  - Primary key: (Matrikel, Lecture ID)

Normalization [10]: My Simplified Perspective

- Normalization: process of organizing tables to minimize redundancy[19]
  - Reduces dependencies within and across tables
  - Prevents inconsistency across replicated information
  - Normally, **reduces required storage space** and **speeds up updates**
- There are different normal forms with increasing requirements
  - 1NF: It follows our notion of a table.
    - No collections in the table. A primary key exists.
  - 2NF: No redundancy of data
    - i.e., entities of many-to-many relations are stored in separate tables
    - Every column must depend on each candidate key and not a subset
  - 3NF: Columns are not functional dependent to sth. else than a candidate key
  - 4NF: Do not store multiple relationships in one table
- 4NF is a good choice[2] for transactional data processing but not big data

[2] It has been shown that 4NF can always be achieved for relational data

Example for Unnormalized Data

Matrikel	**Name**	Birthday	**Name**
242	Hans	22.04.1955	**[Big Data Analytics, Hochleistungsrechnen]**
245	Fritz	24.05.1995	Hochleistungsrechnen

Not normalized Student and lecture table/relation, contains identical column names and collections. Problematic if we want to update the name of an lecture.

Matrikel	Name	Birthday	Lecture Name
242	Hans	22.04.1955	Big Data Analytics
242	Hans	22.04.1955	**Hochleistungsrechnen**
245	Fritz	24.05.1995	**Hochleistungsrechnen**

Student and lecture table/relation in 1NF, it contains a many-to-many relation. Changing lecture name requires still to touch multiple rows.

Example for Unnormalized Data

Matrikel	Name	Birthday	**Age**
242	Hans	22.04.1955	40
245	Fritz	24.05.1995	20

In 2NF but not 3NF: Age is functional depending on birthday

Matrikel	Attended lecture	Attended seminar
242	BDA	SIW
242	HR	SIW
242	BDA	NTH
242	HR	NTH

In 3NF but not 4NF: Candidate key depends on all three columns

Databases [29]

- **Database**: an organized collection of data
  - Includes layout (schemes), queries, views
  - Database models: Relational, graph, document, ...
- **Database management system (DBMS)**: *software application that interacts with the user, other applications and the database itself to capture and analyze data* [29]
  - Definition, creation, update, querying and administration of databases

DBMS functions for managing databases

- Data definition: Creation, modification of definitions for data organization
- Update: Insertion, modification and deletion of data
- Query/Retrieval: retrieving stored and computing derived data
- Administration: users, security, monitoring, data integrity, recovery

Structured Query Language (SQL) [20]

- Declarative language: specify **what** to achieve and not **how**
- Evolving standard with growing feature set

Language elements

- Statement: instructions to perform, terminate by ;
  - Query: alternative name; usually retrieves/computes data
- Clause: components of statements
- Predicates: conditions limiting the affected rows/columns
- Expressions: produce scalar values or tables
- Operators: compare values, change column names
- Functions: transform/compute values

PostgreSQL [10]

A popular database implementation

- Semantics: ACID support for transactions
  - A transaction is a batch of operations
  - It either fails or succeeds
- Implements majority of SQL:2011 standard
  - Syntax may differ from SQL standard and extensions are provided
- Interactive shell via `psql`

Excerpt of features

- Materialized views (create virtual tables from logical tables)
- Fulltext search
- Regular expression
- Statistics and histograms
- User defined objects (functions, operators)
- Triggers: events upon insert or update statements; may invoke functions
- New versions support semi-structed data in arrays, XML, JSON[3]

[3] See http://www.postgresql.org/docs/9.4/static/arrays.html and .../functions-json.html

Schemas (in Postgres)

Creation of a database and table

```
CREATE ROLE "bigdata" NOSUPERUSER LOGIN PASSWORD 'mybigdata';
CREATE DATABASE bigdata OWNER "bigdata";
```

To connect to the database use `psql -W -U bigdata bigdata`

Create our tables

```
CREATE TABLE students (matrikel INT, name VARCHAR, birthday DATE, PRIMARY KEY(matrikel));
CREATE TABLE lectures (id SERIAL, name VARCHAR, PRIMARY KEY(id));
CREATE TABLE attends  (matrikel INT, lid INT,
  FOREIGN KEY (matrikel) REFERENCES students(matrikel),
  FOREIGN KEY (lid) REFERENCES lectures(id));
--\d <TABLE>  prints the schema
```

Constraints (keeps data clean $\Rightarrow$ data governance)

```
-- minimum length of the name shall be 5
ALTER TABLE students ADD CONSTRAINT length CHECK (char_length(name) > 3);
-- to remove the constraint later: ALTER TABLE students DROP CONSTRAINT length ;
-- minimum age of students should be 10 years
ALTER TABLE students ADD CONSTRAINT age CHECK (extract('year' from age(birthday)) > 10);
-- disallow NULL values in students
ALTER TABLE students ALTER COLUMN birthday SET NOT NULL; -- during CREATE with "birthday DATE NOT NULL"
ALTER TABLE students ALTER COLUMN name SET NOT NULL;
```

Populating the Tables

```
-- Explicit specification of columns, not defined values are NULL
INSERT INTO students (matrikel, name, birthday)
  VALUES (242, 'Hans', '22.04.1955');
-- Insertation of the same name twice could be prevented using a constraint
INSERT INTO students (matrikel, name) VALUES (246, 'Hans');
-- Order is expected to match the columns in the table
INSERT INTO students VALUES (245, 'Fritz', '24.05.1995');
INSERT INTO lectures VALUES (1, 'Big Data Analytics');
INSERT INTO lectures VALUES (2, 'Hochleistungsrechnen');

-- Populate relation
INSERT into attends VALUES(242, 1);
INSERT into attends VALUES(242, 2);
INSERT into attends VALUES(245, 2);

-- Insertations that will fail due to table constraints:
INSERT INTO students (matrikel, name) VALUES (250, 'Hans');
-- ERROR:  null value in column "birthday" violates not-null constraint
INSERT INTO students VALUES (250, 'Hans', '22.04.2009');
-- ERROR:  new row for relation "students" violates check constraint "age"
INSERT INTO students VALUES (245, 'Fritz', '24.05.1995');
-- ERROR:  duplicate key value violates unique constraint "students_pkey"
-- DETAIL:  Key (matrikel)=(245) already exists.
```

Queries [20]

- A query retrieves/computes a (sub)table from tables
  - It does not change/mutate any content of existing tables
- Statement: SELECT $< column1 >, < column2 >, ...$
- Subqueries: nesting of queries is possible to create tmp tables

Supported clauses

- FROM: specify the table(s) to retrieve data
- WHERE: filter rows returned
- GROUP BY: group rows together that match conditions
- HAVING: filters grouped rows
- ORDER BY: sort the rows

```sql
SELECT Matrikel, Name FROM students WHERE Birthday='22.04.1955';
-- Returns a table with one row:
--  matrikel | name
-- ----------+------
--       242 | Hans
```

More Queries

Ordering of results

```
-- Example comment, alternatively /* */
select * from students
  where (name != 'fritz' and name != 'nena') -- two constraints
  order by name desc; -- descending sorting order
```

Aggregation functions

```
-- There are several aggregate functions such as max, min, sum, avg
select max(birthday) from students;
-- 1995-05-24

-- It is not valid to combine reductions with non-reduced columns e.g.
select matrikel, max(birthday) from students; -- ERROR!
```

Counting the number of students

```
-- Number of students in the table and rename the column to number
SELECT count(*) AS number FROM students;
-- number
--      2
```

Subqueries

A subquery creates a new (virtual) named table to be accessed

Identify the average age

```
-- Identify the min, max, avg age; we create a new table and convert the date
select min(age), avg(age), max(age) from
    -- Here we create the virtual table with the name ageTbl
    (SELECT age(birthday) as age from students) as ageTbl;
--           min              |                 avg                  |          max
-- 20 years 3 mons 30 days | 40 years 4 mons 15 days 12:00:00 | 60 years ...
```

Identify students which are not attending any course

```
-- We use a subquery and comparison with the set
select matrikel from students
where matrikel not in  -- compare a value with entries in a column
  (select matrikel from attends);
```

Subquery expressions: exists, in, some, all, (operators, e.g., $<$) [4]

[4] See http://www.postgresql.org/docs/9.4/static/functions-subquery.html

Grouping of Data

Data can be grouped by one or multiple (virtual) columns
It leads to errors when including non-grouped / non-reduced values

Identify students with the same name and birthday, count them

```
select name, birthday, count(*) from students group by name, birthday;
-- name  |    max     | count
---------+------------+-------
-- Fritz | 1995-05-24 |     1
-- Hans  | 1955-04-22 |     1
```

Figure out the number of people starting with the same letter

```
select upper(substr(name,1,1)) as firstletter, count(*) from students
   group by firstletter;
-- firstletter | count
---------------+------------
-- F           | 1
-- H           | 1
```

Filtering Groups of Data

- With the HAVING clause, groups can be filtered
- ORDER BY is the last clause and can be applied to aggregates

Identify students with the same name and birthday, and return the total number of non-“duplicates”

```
select sum(mcount) from
  (select count(*) as mcount from students
   group by name, birthday having count(*) = 1 order by count(*)) as groupCount;
-- sum
-- 2

-- Alternatively in a subquery you can use:
select sum(count) from
  (select count(*) as count from students
   group by name, birthday) as groupCount
   where count = 1;
```

Joins [10]

A join combines records from multiple tables

- Used to resolve relations of entities in normalized schemes
- Usually filtering tuples according to a condition during this process

Types of joins

- CROSS JOIN: Cartesian product of two tables (all combination of rows)
- NATURAL JOIN: All combinations that are equal on their common attributes (i.e, both tables contain the matrikel column)
- INNER JOIN: Return all rows that have matching records based on a condition
- OUTER JOIN: Return all rows of both tables even if they are not matching the condition
  - LEFT OUTER JOIN: Return all combinations and all tuples from the left table
  - RIGHT OUTER JOIN: ... from the right table
  - FULL OUTER JOIN: Return all combinations

Example Joins

```
select * from students as s1 CROSS JOIN students as s2;
-- matrikel | name  |  birthday   | matrikel | name  |  birthday
------------+-------+------------+----------+-------+------------
--      242 | Hans  | 1955-04-22 |      242 | Hans  | 1955-04-22
--      242 | Hans  | 1955-04-22 |      245 | Fritz | 1995-05-24
--      245 | Fritz | 1995-05-24 |      242 | Hans  | 1955-04-22
--      245 | Fritz | 1995-05-24 |      245 | Fritz | 1995-05-24

select * from students  NATURAL JOIN attends;
-- matrikel | name  |  birthday   | lid
-- ----------+-------+------------+-----
--      242 | Hans  | 1955-04-22 |   1
--      242 | Hans  | 1955-04-22 |   2
--      245 | Fritz | 1995-05-24 |   2

select * from students INNER JOIN attends ON students.matrikel = attends.matrikel;
-- matrikel | name  |  birthday   | matrikel | lid
------------+-------+------------+----------+-----
--      242 | Hans  | 1955-04-22 |      242 |   1
--      242 | Hans  | 1955-04-22 |      242 |   2
--      245 | Fritz | 1995-05-24 |      245 |   2
```

Example Joins

```
-- This join returns NULL values for Fritz as he has not the selected matrikel
select * from students LEFT OUTER JOIN attends ON students.matrikel = 242;
-- matrikel | name  |  birthday   | matrikel | lid
----------+-------+-----------+---------+-----
--      242 | Hans  | 1955-04-22 |      242 |   1
--      242 | Hans  | 1955-04-22 |      242 |   2
--      242 | Hans  | 1955-04-22 |      245 |   2
--      245 | Fritz | 1995-05-24 |          |

select * from students as s FULL OUTER JOIN attends as a ON s.matrikel = a.lid;
-- matrikel | name  |  birthday   | matrikel | lid
------------+-------+------------+----------+-----
--          |       |            |      242 |   1
--          |       |            |      242 |   2
--          |       |            |      245 |   2
--      242 | Hans  | 1955-04-22 |          |
--      245 | Fritz | 1995-05-24 |          |

-- Now identify all lectures attended by Hans
select s.name, l.name from students as s INNER JOIN attends as a ON s.matrikel
    ↪ = a.matrikel INNER JOIN lectures as l ON a.lid=l.id;
-- name  |         name
---------+---------------------
-- Hans  | Big Data Analytics
-- Hans  | Hochleistungsrechnen
-- Fritz | Hochleistungsrechnen
```

Updating Rows

- UPDATE statement changes values
- DELETE statement removes rows
- Each operation yields the ACID semantics[5]
- Transactions allow to batch operations together

```sql
-- Change the name of Fritz
UPDATE students SET name='Fritzchen' WHERE matrikel=245;

-- Remove Fritzchens attendance in Hochleistungsrechnen
DELETE FROM attends WHERE matrikel=242 and lid=2;

-- Subqueries can be used to select rows that are updated/deleted
-- Remove Fritzchen attendence with the name
DELETE from attends WHERE matrikel=242 and lid = (SELECT id from lectures where name =
    'Hochleistungsrechnen');
```

[5] In fact, when AUTOCOMMIT is enabled, every statement is wrapped in a transaction. To change this behavior on the shell, invoke: SET AUTOCOMMIT [OFF|ON]

Transactions

- Transaction: A sequence of operations executed with ACID semantics
  - It either succeeds and becomes visible and durable; or it fails
  - Note: Complex data dependencies of concurrent operations may create a unresolvable state that require restart
- All queries access data in the version when it started
  - The isolation level can be relaxed, e.g., to see uncommited changes
- Internally, complex locking schemes ensure conflict detection

Example: Atomic money transfer between bank accounts

```sql
START TRANSACTION;
UPDATE account SET balance=balance-1000.40 WHERE account=4711;
UPDATE account SET balance=balance+1000.40 WHERE account=5522;

-- if anything failed, revert to the original state
IF ERRORS=0 COMMIT; -- make the changes durable
IF ERRORS!=0 ROLLBACK; -- revert
```

Performance Aspects

Problem: When searching for a variable with a condition, e.g., x=y, the table data needs to be read completely (full scan)

Indexes

- Index allows lookup of rows for which a condition (likely) holds
- Postgres supports B-tree, hash, GiST, SP-GiST and GIN indexes[6]

```
CREATE INDEX ON students (name);
```

Optimizing the execution of operations (query plan)

- Postgres uses several methods to optimize the query plan
- The query planer utilizes statistics about access costs
  - Knowing how values are distributed helps optimizing access
- ANALYZE statement triggers collection of statistics
- Alternatively: automatically collect statistics
- EXPLAIN statement: describes the query plan (for debugging)

[6] See http://www.postgresql.org/docs/9.4/static/sql-createindex.html

Performance Aspects (2) [22]

Bulk Loads/Restores

- Combine several INSERTS into one transaction
- Perform periodic commits
- Create indexes/foreign key/constraints after data was inserted

Garbage cleaning / vacuuming: Cleaning empty space

- When changing or inserting rows additional space is needed
- It is expensive to identify empty rows and compact them
  - ⇒ Just append new data
  - Mark data, e.g., in a bitmap as outdated
- Periodically space is reclaimed and data structures are cleaned
- VACCUUM statement also triggers cleanup
- ANALYZE also estimates the amount of garbage to optimize queries

Views

- View: virtual table based on a query
  - Can be used to re-compute complex dependencies/apply joins
  - The query is evaluated at runtime, which may be costly
- Materialized view: copies data when it is created/updated[7]
  - Better performance for complex queries
  - Suitable for data analytics of data analysts
  - Export views with permissions and reduce knowledge of schema

```
CREATE  VIEW studentsView AS
  SELECT s.matrikel, s.name as studentName, l.name as lectureName, age(birthday) as age
     ↪ from students as s INNER JOIN attends as a ON s.matrikel = a.matrikel INNER
     ↪ JOIN lectures as l ON a.lid=l.id;

select * from studentsView;
-- matrikel | studentname |     lecturename      |           age
------------+-------------+----------------------+-------------------------
--      242 | Hans        | Big Data Analytics   | 60 years 5 mons 1 day
--      242 | Hans        | Hochleistungsrechnen | 60 years 5 mons 1 day
--      245 | Fritz       | Hochleistungsrechnen | 20 years 3 mons 30 days
-- To replace the data with new data
REFRESH MATERIALIZED VIEW studentsView;
```

[7] www.postgresql.org/docs/9.4/static/sql-creatematerializedview.html

Regular Expressions

- PostgreSQL supports several styles of regular expressions[8]
- We will look at POSIX regular expressions (regex)
- Operator: ~for matching and ~* for not matching
- `regexp_matches(string, pattern)` returns text array with all matches

Examples

```
-- Any lecture which name contains Data
select name from lectures where name~*'data';
-- Big Data Analytics

-- Lectures starting with Big
select name from lectures where name~'^Big.*$';
-- Big Data Analytics

-- Students whose name contain at least two vocals
select name from students where name~'(i|a|o|u).*(a|i|o|u)';

-- Students whose name contain at least one vacal and at most three
select name from students where name~'^([^auiu]*(i|a|o|u)[^aiou]*){1,3}$';

-- Retrieve all lower case letters in the names
select regexp_matches(name, '[a-z]', 'g') as letter from students;
-- {a}, {n} ...
```

Array Operations

- Operations allow manipulation of multidimensional arrays[9]
- Useful operators: `unnest`, `array_agg`, `array_length`
- JSON support in new postgres version (not discussed here)

```sql
-- Alternative schema for our student/lecture example using an array for the attends relationship
CREATE TABLE studentsA (matrikel INT, name VARCHAR, birthday DATE, attends INT[], PRIMARY KEY(matrikel));
CREATE TABLE lectures (id SERIAL, name VARCHAR, PRIMARY KEY(id));

INSERT INTO studentsA VALUES (242, 'Hans', '22.04.1955', '{1,2}' );
INSERT INTO studentsA VALUES (245, 'Fritz', '24.05.1995', '{2}');

-- Addressing array elements: first lecture attended by each student
SELECT attends[1] from studentsA;
-- Slicing is supported: First three lectures
SELECT attends[1:3] from studentsA;

-- Retrieve the lecture name attended for each student
SELECT s.name, l.name from studentsA AS s INNER JOIN lectures AS l ON l.id = ANY(s.attends);
-- Hans  | Big Data Analytics
-- Hans  | Hochleistungsrechnen
-- Fritz | Hochleistungsrechnen

-- Now retrieve the lectures in an array per person
SELECT s.name, array_agg(l.name) from studentsA AS s INNER JOIN lectures AS l ON l.id = ANY(s.attends)  GROUP by s.matrikel;
-- Hans  | {"Big Data Analytics",Hochleistungsrechnen}
-- Fritz | {Hochleistungsrechnen}
```

[9] See http://www.postgresql.org/docs/9.4/static/arrays.html

Processing Geospatial Data with PostGIS [30, 31]

- PostGIS is a PostgreSQL extension providing datatapes and functions for
  - Topology: Faces, Edges and Nodes
    - Defines constraints on data, e.g., sharing of edges in maps
  - Geometry/Geography: coordinates according to SRID
    - Spatial Reference System Identifier (SRID) defines coordinate system
    - Lon/Lat coordinates on a sphere with the unit degrees
    - Points, lines, poligones
  - Raster data: like pixels, square-based split of a 2D plane
    - Example: Import / export of images
- QGIS viewer[10] can visualize geometry and raster data

[10] http://qgis.org/

PostGIS: Example [31]

```
-- Creating a database with geography data (SRID 4326 => WGS 84 => for GPS => lon/lat)
CREATE TABLE cities(gid serial PRIMARY KEY, n TEXT, loc geography(POINT,4326) );
CREATE INDEX cities_idx ON cities USING GIST ( loc );

-- Insert three cities with Lon/Lat coordinates
INSERT INTO cities (n, loc) VALUES('Hamburg',ST_GeographyFromText('POINT(9.99 53.5)'));
INSERT INTO cities (n, loc) VALUES('Tokio',ST_GeographyFromText('POINT(139.8 35.65)'));
INSERT INTO cities (n, loc) VALUES('Aleppo',ST_GeographyFromText('POINT(37 36)'));

-- Compute distance between Hamburg and Tokio
SELECT ST_Distance( (Select loc from cities where n = 'Hamburg'),
                    (Select loc from cities where n = 'Tokio'));
-- 9012369.89691784 == 9012 km

-- How far is Allepo from a plane flying from Hamburg to Tokio, here as text
SELECT ST_Distance('LINESTRING(9.99 53.5, 139.8 35.65)'::geography,
      'POINT(37 36)':: geography);
-- 2833 km
```

Data Warehouse

"A data warehouse (DW or DWH), also known as an enterprise data warehouse (EDW), is a system used for reporting and data analysis." [27]

- Central repository
- Integrates data from multiple inhomogeneous sources
- Data analysts use a simplified data model: a multidimensional data cube
- Provides tools for the data analyst to support descriptive analysis
- May provide some tools for predictive analysis
- Many queries are executed periodically and used in reports

Databases vs. Data Warehouses for Structured Data

Database management systems (DBMS)

- Standardized systems and methods to process structured data
- Use the relational model for data representation
- Use SQL for processing

Online Transaction Processing (OLTP)

- Real-time processing
- Offer ACID qualities
- Relies on normalized schemes (avoid redundant information)

Online Analytical Processing (OLAP)

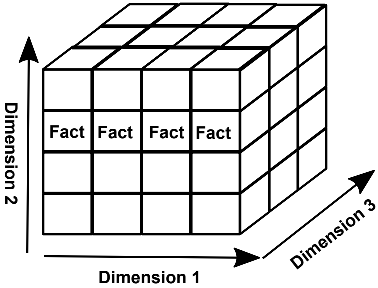
- Systems and methods to analyze large quantities of data
- Utilizes data warehouses with non-normalized schemes
- Extract, Transform and Load (ETL): import data from OLTP

OLAP

- Online analytical process with large quantities of business data
- Utilizes denormalized dimensional model to avoid costly joins
- Technology alternatives:
  - **MOLAP** (Multidimensional OLAP): problem-specific solution
  - **ROLAP**: use relational databases to represent cube
    - Star schema
    - Snowflake schema
- **Dimensional modeling**: design techniques and concepts [26]
  1. Choose the business process, e.g., sales situation
  2. Declare the grain: what does the model focus on, e.g., item purchased
  3. Identify the dimensions
  4. Identify the facts

The OLAP Cube: Typical Operations [27]

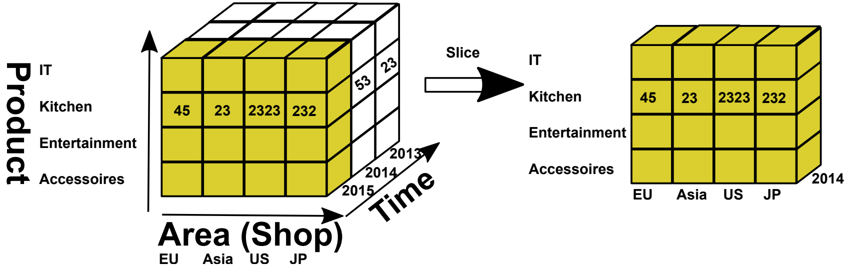
- Slice: Fix one value to reduce the dimension by one
- Dice: Pick specific values of multiple dimensions
- Roll-up: Summarize data along a dimension
  - Formulas can be applied, e.g., profit = income - expense
- Pivot: Rotate the cube to see the faces



Example 3D cube

The OLAP Cube: Slice [27]

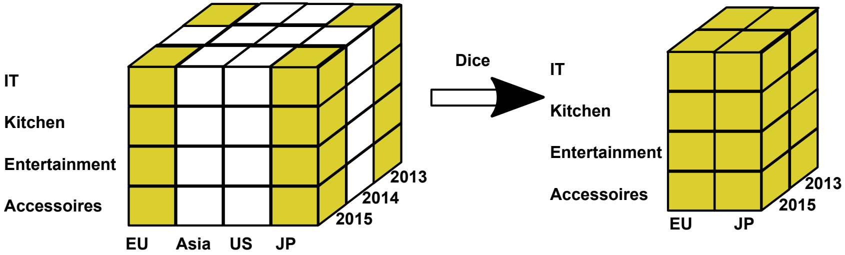
- Slice: Fix one value to reduce the dimension by one
- Example: Sales (in Euro) for worlwide stores



Example cube for sales in stores

The OLAP Cube: Dice [27]

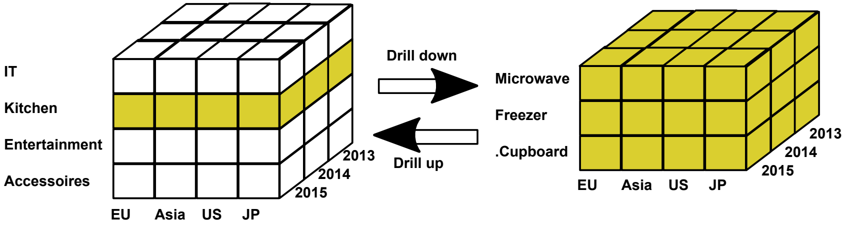
- Dice: Pick specific values of multiple dimensions



Example cube for sales in stores

The OLAP Cube: Drill Down/Up [27]

- Drill Down/Up: Navigate the aggregation level
  - Drill down increases the detail level
  - Drill up decreases the detail level



Example cube for sales in stores

Star (and Snowflake) Schemas [23]

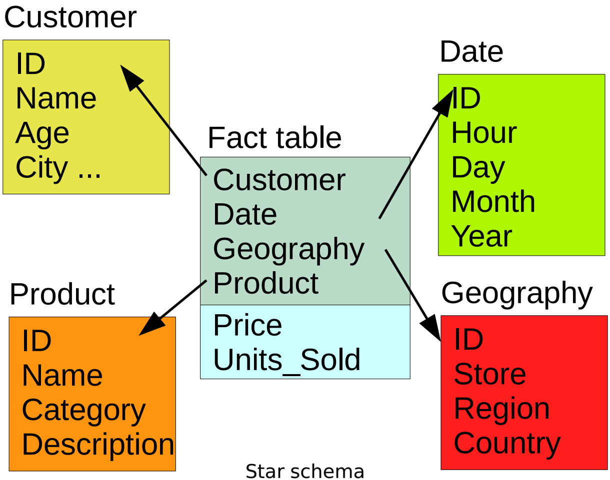
Implement the OLAP cube in relational databases

Data model

- Fact table: records measurements/metrics for a specific event
  - Center of the star
  - Transaction table: records a specific event, e.g., sale
  - Snapshot table: record facts at a given point in time, e.g., account balance at the end of the month
  - Accumulating table: aggregate facts for a timespan, e.g., month-to-date sales for a product
  - ⇒ A fact table retains information at a low granularity and can be huge
- Dimension tables: describe the facts in one dimension
  - Contains, e.g., time, geography, product (hierarchy), employee, range
  - The fact table contains a FOREIGN KEY to all dimension tables
  - ⇒ Comparably small tables

Snowflake schema normalizes dimensions to reduce storage costs

Star Schema Example Model



Star schema

Star Schema: Example Query

Analyze the sales of TVs per country and brand [23]

```sql
SELECT  P.Brand, S.Country AS Countries, SUM(F.Units_Sold)
FROM Fact_Sales F
INNER JOIN Date    D ON (F.Date_Id = D.Id)
INNER JOIN Store   S ON (F.Store_Id = S.Id)
INNER JOIN Product P ON (F.Product_Id = P.Id)

WHERE D.Year = 1997 AND  P.Product_Category = 'tv'

GROUP BY
  P.Brand,
  S.Country
```

Star Schema [23]

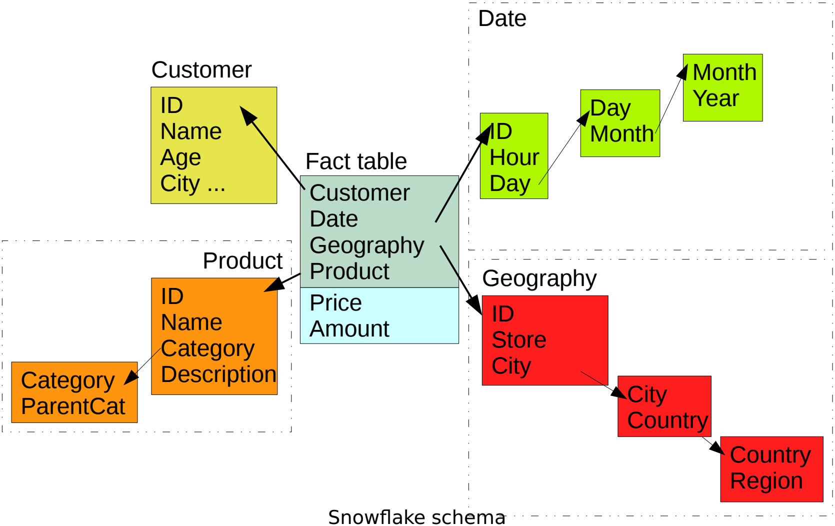
Advantages

- Simplification of queries and performance gains
- Emulates OLAP cubes

Disadvantages

- Data integrity is not guaranteed
- No natural support for many-to-many relations

Snowflake Schema Example Model



Snowflake schema

Summary

- ER-diagrams visualize the relational data model
- Keys allow addressing of tuples (rows)
- Normalization reduces dependencies
  - Avoids redundancy, prevents inconsistency
- SQL combines data retrieval/modification and computation
  - Insert, Select, Update, Delete
  - Joins combine records
- Transactions executes a sequence of operations with ACID semantics
- A database optimizes the execution of the queries (query planer)
- Semi-structured data analysis is possible within JSON and XML
- OLAP (Cube) deals with multidimensional business data
- Data warehouses store facts along their dimensions
- Star-schema implements OLAP in a relational schema

Bibliography

10 Wikipedia
11 https://en.wikipedia.org/wiki/Relational_model
16 https://en.wikipedia.org/wiki/Superkey
17 https://en.wikipedia.org/wiki/Candidate_key
18 https://en.wikipedia.org/wiki/Unique_key
19 https://en.wikipedia.org/wiki/Database_normalization
20 https://en.wikipedia.org/wiki/SQL
21 PostgreSQL Documentation http://www.postgresql.org/docs/9.4/static/
22 https://wiki.postgresql.org/wiki/Performance_Optimization
23 https://en.wikipedia.org/wiki/Star_schema
24 https://en.wikipedia.org/wiki/Data_mart
25 https://en.wikipedia.org/wiki/Snowflake_schema
26 https://en.wikipedia.org/wiki/Dimensional_modeling
27 https://en.wikipedia.org/wiki/OLAP_cube
28 https://en.wikipedia.org/wiki/Data_warehouse
29 https://en.wikipedia.org/wiki/Database
30 http://www.bostongis.com/?content_name=postgis_tut01
31 http://postgis.net/docs/manual-dev/