Physical data modelling

Mullins chapter 4

Bjarte Wang-Kileng

HVL

February 13, 2025

Western Norway University of Applied Sciences

Outline

Outline

Moving from a conceptual- to a logical model

- Normalise.
- Specify the data types or domains of the attributes.
- Specify candidate keys and PK.
- Specify nullability of the attributes.
- Remove many to many relationships.

Moving from a logical- to a physical model

- Entities are mapped to tables.
- Attributes are mapped to columns.
- Handle the domains of the attributes.
- Indexes.
- Denormalise?

Handle the domains of the attributes

- Each column must be assigned a data type supported by the DBMS.
    - SQL compatibility table
- Multiple physical data types may be candidates for a column:
    - The year of birth of a person may be TINYINT, SMALLINT, MEDIUMINT, YEAR, etc.
- Must determine:
    - What data type can e.g. be most efficiently accessed, stored and maintained?
    - How will the data be used by the applications?

Handle the domains of the attributes, contd.

- Constraints on columns can place limits on data:
  - Column *year_born* should be larger than 1900, but less than 2100.
  - Can be implementeted using the CHECK clause or triggers.
- Default values of columns.
- Nullability – should NULL values be allowed?
- Text and character data – fixed or variable length?
  - Variable length can save space, but
  - performance can suffer.

The primary key

- PK of the physical model can be different from the logical model.
  - The DBMS might not allow indexes on certain data types.
  - The PK of the logical model is long.
  - The PK of the logical model involves many columns.
- Choose the primary key columns carefully based on the most performance-critical queries.
- The PK should never (or rarely) change.

DBMS generated primary key values

- DBMSs have built in features for automatic creating PK values.
  - MySQL and MariaDB: AUTOINCREMENT
  - IMB DB2, Microsoft SQL server: IDENTITY(1,1)
  - Derby, DB2, Oracle: GENERATED AS IDENTITY
  - Oracle: CREATE SEQUENCE
- Can be used if there is no good PK candidate.

The primary key

MySQL/MariaDB and the InnoDB/XtraDB engine

- Long primary keys wast a lot of disk space.
- MariaDB – see e.g. Getting Started with Indexes.

Column ordering

- In the logical model, the order of the columns is irrelevant.
- In the physical model, the order can influence performance.
- E.g. DB2 and logging of changes:
  - Writes to log from first till last byte changed.
  - For variable length row, writes to log from first byte changed to the end of the row.
- Least logging of changes also but fastest retrieval of data:
  - Frequently updated columns are grouped together last in row.

Referential integrity[1]

- Know from the course *Databaser*.
- Implement referential integrity using DMBS constraints, rather than trusting the applications.
- What happens with the FK (*Foreign Key*) when a PK (*Primary Key*) is delete?
  - Need rules.

[1] Covered in depth later in the course.

Referential integrity in MariaDB

- XtraDB and InnoDB enforces foreign key constraints.
- InnoDB is the default engine in MySQL and MariaDB.

InnoDB foreign key referenial integrety

- `ON DELETE RESTRICT/NO ACTION` and `ON UPDATE RESTRICT/NO ACTION`.
- `ON DELETE CASCADE` and `ON UPDATE CASCADE`.
- `ON DELETE SET NULL` and `ON UPDATE SET NULL`.

Physical files and Tablespaces

- Tablespaces/data spaces – the physical structures that stores the database objects.
  - Remember to include space for indexes, log files etc.
- DBMS stores the tablespace in physical files.
- Each tablespace resides in one more files.
- Each database can cover one or more tablespaces.

Tablespaces and MySQL/MariaDB

- A tablespace can be spread across different file systems to improve performance, i.e. *partitioning*.
  - A table must use the same engine on all file systems, or *partitions*.
- For each database, a folder is created with the name of the database.
  - The folder stores one or more files for each table of the database.
  - Regardless of storage engine, a file **TableName.frm** stores the data table format.
- Case sensitivity of database- and table names depend on the case sensitivity of the file system.
  - Case insensitive on Windows.
  - Case sensitive on Unix and Linux.
  - Mac OS X is Unix, but the default file system HFS+ is case insensitive.

Tablespace for MyISAM and Aria (MySQL/MariaDB)

- Each database is stored in a folder with the name of the database.
- One tablespace for each table.
- Each tablespace is stored in 3 files with names given by the table.
  - MyISAM:
    - **TableName.MYD** – table data.
    - **TableName.frm** – table format.
    - **TableName.MYI** – index file.
  - Aria:
    - **TableName.MAD** – table data.
    - **TableName.frm** – table format.
    - **TableName.MAI** – index file.

Tablespace for NDB cluster (MySQL only)

- Each tablespace contains one or more data files and log files.
- Multiple tablespaces can be declared, see section CREATE TABLESPACE Syntax of the MySQL manual.
- Not supported on MariaDB.

Tablespace for InnoDB

- Default is to place every table in its own tablespace.
  - Option *innodb_file_per_table*
- Can also use a single tablespace for all databases.
  - Stores data and indexes for all InnoDB tables across all databases.
- A single tablespace can be spread on several files and disks, e.g.:

```
innodb_data_file_path=/mnt/disk1/ibdata1:50M;/mnt/disk2/ibdata2:50M:autoextend:max:500M
```

- An unmounted raw disk partition can be used for the tablespace:
  - Will improve performance.

Outline

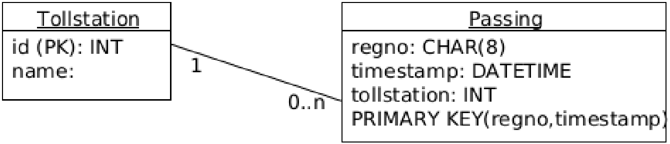
Introduction to indexes

- Know from the course *Databaser*.
- An index gives an alternative path to data:
  - Fewer I/O operations to find data.
  - Faster queries.
  - Some queries can involve only the index tree.
- The DBMS, not the programmer determines wheter to use an index.
- In order to design good indexes, the DBA must:
  - Determine what types of queries will be run,
  - understand the DBMS relational optimiser.

Introduction, contd.

- Design of indexes will be covered more extensively later in the course.
- No appropriate index will most likely give a *table scan*:
  - Search through all rows of table.
  - Worst type of search when it comes to performance.
- `EXPLAIN` can show how indexes are used by a `SELECT`.
- `ANALYZE TABLE` command in MySQL/MariaDB analyses and stores the index distribution for a table.

Demo – EXPLAIN



- `EXPLAIN SELECT ...`
- The DBMS will read a row from the first table listed and match it with a row in the next table listed repeated for all rows of the first table.
- See e.g. EXPLAIN - MariaDB Knowledge Base.

Demo – EXPLAIN

Examples of output field values

- id: The sequential number of the SELECT within the query.
- select_type: **SIMPLE** – no UNION or sub queries.
- type, e.g.:
    - **index** – full scan through all indexes.
    - **ref** – all rows with matching indexes are read for each row of the corresponding table.
    - **ALL** – full table scan.
- possible_keys: Lists keys (indexes) that the DBMS can choose.
- key: Show what index was used.
- ref: Show what column that was compared to the index of *key*.
- rows: Estimate for the number of rows the DBMS must search.
- Extra: **Using index** – only the index tree must be read, not the table.

When to make indexes?

- Large tables.
- Frequently accessed tables.
- For queries that access few rows (less than 25%).
  - If more rows are selected, table scans will usually be as good.
- Always use index for the PK.
- Also use index for FK – better performance on enforcing the referential integrity.
- Candidate keys – use indexes for candidate keys if used when looking up data.
  - Use the UNIQUE constraint if MariaDB.

Indexes and columns

- Create indexes on the most-referenced columns.
- Create indexes on columns to minimize sorting:
  - JOINs.
  - ORDER BY.
  - GROUP BY.
  - UNION.
  - DISTINCT.
- If most queries involve the same subset of columns, a composite index involving these columns will be good:
  - A single composite index can be used even if some queries involve a subset of the index-colums → more later
  - Column order can be important.

Queries

Indexes and queries

Design of proper indexes require knowledge of how tables are to be accessed.

Carefully study the potential queries.

Drawbacks when using indexes

- Additional overhead when rows are inserted, deleted or updated:
  - Indexes are best for data that is mostly static.
- Additional disk space is required.
- An index good for some queries might adversely affect performance of other queries.
- A more complex database might be more difficult to maintain.

Tablespace and cache

- Separate caches for data and index reads can save space:
  - Index entries are smaller than a full table row.
  - More index entries can fit into the cache.
- Storing indexes and table data on separate disks can minimise disk seek time.

Types of indexes

- B-tree (or B+-tree) – default index structure for all major DBMSs.
- Bitmap.
- Reverse key index.
- Partitioned indexes.
- Clustered indexes. (Obs.: Has **nothing** to do with DBMS clusters.)
- Hashing.

B-tree

- Root node – here the search starts.
- Leaf nodes – lowest level of nodes. Pointers to the actual data.
- Distance from root node to leaf is the the same for all leaf nodes.
  - Distance do vary with time as index entries are added or deleted.

B-tree example

NAME	PHONE	YEAR_BORN	SEX
SAM	34879778	22	M
DAVID	56784457	32	M
SUSAN	34465567	22	F
BARRY	12345678	21	M
MARK	23780965	25	M
SID	23656778	26	M
CARLA	65467821	20	F
OTTO	46745455	34	M
JOE	54567676	29	M
SUE	45676877	19	F

Table: Table *FRIENDS*

- Column **NAME** is an index

B-tree example – using the index file

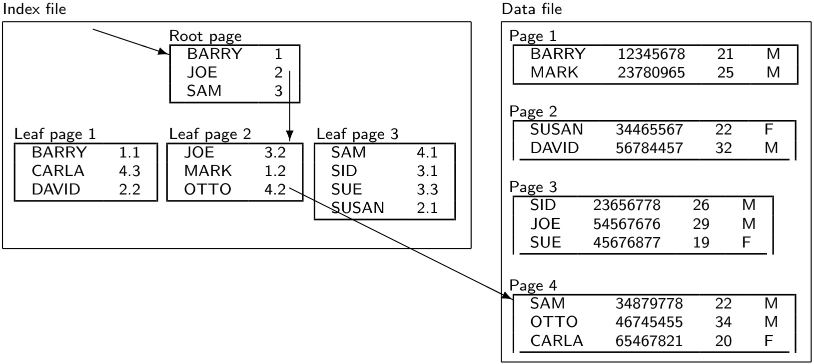


Figure: SELECT * FROM FRIENDS WHERE NAME = 'OTTO'

Bitmap

- Infrequently modified data.
- Columns with small domains, eg. boolean, gender etc.
- Assume the following bitmaps for gender:

  Male: 10010
  Female: 01001
  Unknown: 00100

  Rows 1 and 4 are **male**, rows 2 and 5 are **female** and row 3 is **unknown**.

Reverse key index

- Can improve B-tree distribution.
- The index is based on the reverse of the column data.
- Adjacent key values will not be stored together.
- For *FirstName* «Grete», the index will be based on «eterG».
- Good for direct data look up, but bad when retrieving ranges of data.
- Oracle, but not MySQL or MariaDB.

Partitioned indexes

- B-tree indexes with data spread across multiple partitions.
- A database partition is a division of DB objects (e.g. tables) into distinct independent parts.
  - Different partitions can be stored on different disks or nodes in a DB cluster.
- Can enhance performance and availability.
- See e.g. KEY Partitioning (MySQL) or Partitioning Types (MariaDB).

Clustered indexes

- Data table is stored sorted on disk.
- Rows are sorted on disk with respect to the clustered index.
- Why – Data that is commonly accessed together is stored together on same or contiguous pages.
- Increased performance – fewer I/O requests needed when retrieving sequences of data.
- A table can only have one clustered index. Why?

When to use clustered indexes

- Large number of queries retrieve ranges of data based on specific column values.
- A foreign key can be a good candidate. The foreign key typically is the "many" end of a one-to-many relationship.
- When data is frequently sorted (ORDER BY, GROUP BY, etc.).
- Data should be rather static. Inserts and updates can reduce the clustering.
- The PK is normally not a good clustering index.

Clustered indexes in MySQL and MariaDB

- With engines InnoDB.
- There will always be one clustered index for an InnoDB table.
- If a PK exists – InnoDB will always use the PK as a clustered index.
- See e.g. Clustered and Secondary Indexes (InnoDB).

Hashing

- Uses a hash function to spread the key values throughout the physical storage.
- Good for direct data look up, but bad when retrieving ranges of data.
  - The hash function gives a pointer to the physical location of the row.
- Normally only one I/O request when retreiving a row of data.
- MySQL/MariaDB:
  - With engines *Memory* and *NDB*.
  - InnoDB using an Adaptive Hash Index (MySQL, MariaDB).

Interleaving data

- If two tables are joined frequently, the two tables can be stored together in the same physical storage.
- The data is interleaved on disk based on the join criteria.
- Join performance may improve, but only for the specific join.
- Oracle uses clustering to support interleaving.
- Other DBMSs might need to interleave data before loading.

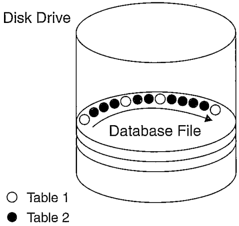


Figure 4.3 *Interleaving table data*

Outline

Normalised data – why or why not?

- Each fact is stored in only one place.
- Retrieving different but related facts require many look ups. Might be slow.
- Updating is fast. Only one place to update.

Denormalisation

Denormalisation

Introduce redundancies to speed up retrieval of data.

Denormalisation and updates

When the same fact is stored several places, updates and inserts will be slower and more complicated.

When to denormalise

- Never denormalise unless required due to performance issues.
- DBMS issues might influence the decision.

Factors to consider before denormalising

- Can the system achieve acceptable performance without denormalising?
- Will the performance after denormalising still be unacceptable?
- Will the system be less reliable due to denormalisation?

Situations where denormalisation can improve performance

- Joins – joins are expensive.
- Repeating groups.
- Calculations on columns are necessary to retrieve data:
  - Precalculate and store the answer in a table.
- Concurrent but different access by different users.
- Large primary keys:
  - Can waste a lot of disk space when carried as foreign key columns in related tables.
  - Can waste a lot of disk space with InnoDB since the primary key is stored with every secondary index.
- Certain columns are queried a large percentage of the time.

Two sets of tables

- If enough disk space, use two sets of tables.
- One set is fully normalised.
- One set is denormalised.
- Updates are done in the normalised tables.
- Reads are done from the denormalised tables.

Two sets of tables

Synchronise the denormalised tables

- If reading does not need to be completely up to date, run updating programs at specified intervals.
- Can use triggers, or allow updates only through procedures.
- Might have to program the applications to keep data consistent.

Single table set

- If not enough disk space, use only the denormalised table set.
- Can use triggers to keep redundant data consistent.a
- Can require procedues stored in the DBMS to update data.
  - The procedures must be programmed to keep data consistent.
- Might have to program the applications to keep data consistent.
  - Better to use procedures stored in the DBMS.

Use of denormalisation

- Program the applications so that a switch to normalised data is easy.
- A new version of the DBMS might work well with normalised data.
- The data access layer of the applications might have to be programmed to keep data consistent.
- Document every denormalisation decision:
  - The reason for the denormalisation decision.
  - The exact changes from the normalised logical model.

Use of denormalisation, contd.

When to denormalise?

Only denormalise the physical model after concrete evidence for its necessity.

A skilled DBA might know from prior direct experience that a specific application and DBMS will need denormalisation in a specific situation.

Outline

Prejoined Tables

- When to use:
  - If two or more tables need to be joined on a regular basis, and
  - the cost of the join is prohibitive.
- Only include columns necessary for the actual queries done by the applications.
- The prejoined table must be updated when changes are made to the normalised tables.
  - Can create a procedure to insert data, that also updates the prejoined table.

Prejoined tables

Example

- Refer slide 24 in lecture from Mullins ch. 3.
- Consider that applications frequently issue the following SELECT:

```
SELECT Major.StudentMajor FROM Student JOIN Major
  ON Student.MajorID=Major.MajorID AND Student.StudentID=?
```

- Then a prejoined table can be made:

StudentID	StudentMajor
12	Informatics
14	Informatics
17	Elkraft

Table: A prejoined table *STUDENT_AND_MAJOR*

- Now, the following SELECT will find the same data:

```
SELECT StudentMajor FROM STUDENT_AND_MAJOR WHERE StudentID=?
```

Combined tables

- Tables with a one-to-one relationship can be combined into one table.
- Sometimes also tables with a one-to-many relationship can be combined:
  - Data update process will be more complicated.
- Similar to the prejoining example, but we keep all columns (except perhaps a redundant FK).

Combined tables

- Refer slide 24 from lecture on Mullins ch. 3.

StudentID	LastName	FirstName	MajorID	StudentMajor
12	Olsen	Ole	INF	Informatics
14	Annesen	Anne	INF	Informatics
17	Gretesen	Grete	EL	Elkraft

Table: STUDENT_AND_MAJOR

Report tables

- E.g. used by data warehouses.
- Reports using data from a database can need special formatting and/or data manipulation by applications.
- The report can be stored in the database and retrieved by SQL.

Mirror tables

- The load on heavily accessed tables can be eased by making duplicates of the tables.
- Can be used if some traffic only read data, and the data need not be completely up to date:
  - E.g data warehouse and decision making can often succeed with nearly correct data.
- Need mechanisms for synchronising data:
  - Batch jobs.
  - Mechanisms can sometimes built into the DBMS (replication).
- Physical implementation may be rather different for the different versions of the tables:
  - Different queries can cause different decisions concerning e.g. indexes.

Split tables

- Some queries use some pieces of a table, other queries use other pieces of the same table.
- No or few queries use both pieces.
- Consider then to split the table, one for each group of queries.
- If some queries use both pieces of the table:
  - Keep a copy of the complete table, or
  - create a *view* joining the tables.
- A table can be split both vertically and horizontally.

Split tables

Vertical split

- The table is replaced by two tables.
- One table has some of the columns, the other has the rest of the columns.
- The PK columns are included in both tables.
- Keep all rows in both tables:
  - Updates and retrieval will otherwise be complicated.

Split tables

Horizontal split

- The table is replaced with two tables having the same columns.
- The rows of the original table is distributed between the two new tables.
- Usually, the split is done using the PK:
  - Some PK ranges is placed in one table, the remaining in the other.
  - E.g.: *LastName* starting with [A-H] in one table, [I-Z] in the other.
- Never have the same row in both tables.

Redundant data

- If queries from one table (*TableA*) usually result in queries to some columns of another table, add these columns to the first table, *TableA*.
- The added columns will then be redundant data in *TableA*.
- Joins can be eliminated and performance increased.
- Do not remove the columns, make a copy.
- Only add redundant columns if:
  - The number of redundant columns are small.
  - The redundant data is rather static.

Repeating Groups

- Repeating groups in data:
  - Can not be implemented in a relational DBMS.
  - 1NF solution – make a new table for the repeating groups.
- A repeating group can be denormalised into columns:
  - If n is the maximum number of values in the repeating group, then
  - add n columns to the table.
- 1NF solution:
  - Optimises data integrity.
  - Give better update performance.
- Denormalising into columns can give less I/O, and more efficient data retrieval.

Repeating groups

Example

- Refer slide 24 in lecture on Mullins ch. 3.

StudentID	LastName	FirstName	StudentMajor	CourseName1	CourseName2	CourseName3
12	Olsen	Ole	Informatics	TOD062	TOD072	
14	Annesen	Anne	Informatics	TOD072	FOA031	FOA052
17	Gretesen	Grete	Elkraft	TOE152	HOE076	

Table: STUDENT_AND_ENROLMENT

Repeating groups by adding columns – requirements

- No aggregation or comparison is done within the repeating group.
- The maximum number of values are stable.
- The data in the repeating group is usually accessed collectively.

Derivable data

- Derivable data is data computed from data in the database.
  - E.g. aggregation (COUNT, SUM, AVG).
- If the derivable data is frequently accessed:
  - Precalculate and store result in columns.
- Need mechanisms to update the derived data when the underlying data changes.
  - Can e.g. create a procedure to insert data that also updates the derivable data.

Derivable data – example

- Customer-table – Add a column *TotalSum* that shows the total sum of everything that this customer has bought.
- Otherwise, calculating the total sum will require a lot of I/O to many different tables.
- See also exercise 3b of TOD122 from the exam of spring 2011.

Criteria for storing derived data

- The source data is rather static.
- The cost of calculating the derived data is quite high.
- The derived data can be synchronised with the underlying data.

DBMS needs

- Requirements of the database and physical details of the DBMS might not mix well for good performance.
- Limitations, requirements and needs of the DBMS might necessitate a denormalised physical model.

DBMS needs

Examples with MySQL/MariaDB

- InnoDB Limits (InnoDB Limitations - MariaDB Knowledge Base):
  - An InnoDB table can contain a maximum of 1017 columns.
  - The maximum row size, except for variable-length columns (VARBINARY, VARCHAR, BLOB and TEXT), is slightly less than half of a database page.

- See also Limits on Table Column Count and Row Size (MySQL).

Speed tables

- Not subject for the exam.
- Storing a pre traversed hierarcial structure.
- See the book for details.

Outline

Views

- A logical table.
- Name for a SELECT.

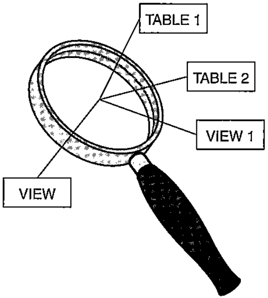


Figure 4-4 *What is a view?*

Example with MySQL/MariaDB:

```
CREATE VIEW PassCar (name,time,regno) AS
    SELECT T.name,P.timestamp,P.regno FROM TollStation T
    JOIN Passing P ON T.id=P.station;
```

Building blocks for Views

- Rows from tables.
- Rows from views.
- Columns from tables.
- Columns from views.

What can we use Views for?

- Provide row and column level security.
- Mask complexity from user.
- Ensure proper data derivation.
- Ensure efficient access paths.
- Rename tables.
- Rename columns.

Outline

Data Definition Language

- Also named SQL data definition language:
  - DDL is a subset of the SQL commands.
  - For other classes of SQL, see the SQL92 standard, section 4.22.2.
- SQL commands used to create, destroy and alter database objects.
- Basic DDL commands: `CREATE`, `ALTER` and `DROP`.

Outline

Temporal Data

- Applications might need to access noncurrent data:
  - Insurance customer now reports a damage that occured 3 months ago.
  - What insurances was valid for this customer 3 months ago?
- A temporal database can access the state of a database as it was at different points in time.
- A traditional, or current database can only store facts that are believed to be true at the current point in time.

Business time and System time

- Business time (valid time, or *application time* in SQL:2011) denotes the time period when a fact is true with respect to the real world.
- System time (transaction time, or *system time* in SQL:2011) is the time when the fact is stored in the database.
- A customer pays his subscription fee for a magazine:
  - Payment is registered on December 1, 2022 – system time.
  - The subscription is valid for 2023 – business time.
- *Bitemporal support* – Both business- and system temporal data support.

Support for temporal tables

- SQL:2011 has some temporal extensions.
- A DBMS with temporal support must attach a time period to data telling when it is valid and when it changed.
  - DB2, PostgreSQL, Oracle, Microsoft SQL Server, MariaDB.
  - Not yet MySQL.
- Without DBMS support, different approaches are:
  - Separate history tables updated by triggers,
  - snapshot tables,
  - adding temporal columns to data tables and queries.
- MariaDB temporal tables:
  - System-Versioned Tables (for sytem time).
  - Application-Time Periods (for business time).
  - Bitemporal Tables
- More details later.