IF3140 – Sistem Basis Data

# Schema and Index Tuning

SEMESTER I TAHUN AJARAN 2024/2025

KNOWLEDGE & SOFTWARE ENGINEERING

Modified from Silberschatz's slides, "Database System Concepts", 7th ed.

## Sources

- “Database Systems: Design, Implementation, and Management”, ninth edition, by Coronel, Morris, Rob; chapter 11 (Cengage Learning, 2010)
- “Database System Concepts”, seventh edition, Silberschatz, Korth, Sudarshan, chapter 25 (McGraw Hill, 2019)
- Other sources

## Objectives

- Students are able to perform higher level database design tuning (i.e. schema tuning, materialized view, and index tuning) in order to improve the database performance

KNOWLEDGE & SOFTWARE ENGINEERING

## Higher-Level Database Design Tuning

- After ER design, schema refinement, and the definition of views, we have the **conceptual** and **external** schemas for our database.
- The next step is to choose **indexes** and to **refine** the conceptual and external schemas (if necessary) to meet performance goals.
- We must begin by understanding the **workload**:
  - The most important queries and how often they arise.
  - The most important updates and how often they arise.
  - The desired performance for these queries and updates.

## Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

## *Decisions to Make*

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered/non-clustered? Hash/B$^+$-tree? Bitmap?
- Should we make changes to the logical schema?
  - Consider alternative normalized schemas? (Remember, there are many choices in decomposing into BCNF, etc.)
  - Should we "undo" some decomposition steps and settle for a lower normal form? (Denormalization)
  - Horizontal partitioning, replication, views, ….

KNOWLEDGE & SOFTWARE ENGINEERING

## *Schema Tuning*

## Schema Tuning (I)

- Choice of logical schema should be guided by workload, in addition to redundancy issues:
  - We might consider **horizontal decompositions.**
  - We might consider **vertical decompositions**.
  - We may settle for a 3NF schema rather than BCNF.
  - Workload may influence choice we make in decomposing a relation into 3NF or BCNF.
  - We may further decompose a BCNF schema!
  - We might **denormalize** (i.e., undo a decomposition step), or we might add fields to a relation.
- If such changes are made after a database in use, called **schema evolution**; might mask changes by defining views

## Schema Tuning (2)

- Can be done in several ways:
  - **Splitting tables**
    - Sometimes splitting normalized tables can improve performance
    - Can split tables in two ways: (1) **Horizontally**; (2) **Vertically**
    - Add complexity to the applications
  - **Denormalization**
    - Adding redundant columns
    - Adding derived attributes
    - Collapsing tables
    - Duplicating tables

## *Schema Tuning: Horizontal Splitting (I)*

- **Horizontal splitting:** Placing rows in two separate tables, depending on data values in one or more columns
- Use horizontal splitting if:
  - A table is large, and reducing its size reduces the number of index pages read in a query
  - The table split corresponds to a natural separation of the rows, such as different geographical sites or historical vs current data
    - Might choose horizontal splitting if a table stores huge amounts of rarely used historical data, and the applications have high performance needs for current data in the same table
  - Table splitting distributes data over the physical media

## Schema Tuning: Horizontal Splitting (2)

### Horizontal Splitting Example

**Problem**:
Usually only active records are processed

| Authors | | |
|---|---|---|
| Active | | |
| Active | | |
| Inactive | | |
| Active | | |
| Inactive | | |
| Inactive | | |

**Solution**: Partition horizontally into active and inactive data

| Active_Authors | | |
|---|---|---|
| Active | | |
| Active | | |
| Active | | |

| Inactive_Authors | | |
|---|---|---|
| Inactive | | |
| Inactive | | |
| Inactive | | |

## Schema Tuning: Vertical Splitting (I)

- **Vertical Splitting:** Partition relations to isolate the data that is accessed most often – only fetch needed information
- Keeps the relation in normal form
- Use vertical splitting if:
  - Some columns are accessed more frequently than other columns
  - The table has wide rows, and splitting the table reduces the number of pages that need to be read
- Makes even more sense when both of the above conditions are true
  - When a table contains very long columns that are accessed infrequently, placing them in a separate table can greatly speed the retrieval of the more frequently used columns
  - With shorter rows, more data rows fit on a data page, so for many queries, fewer pages can be accessed

## Schema Tuning: Vertical Splitting (2)

**Vertical Splitting Example**

- Account relation with the following schema:

account (<u>account-number</u>, branch-name, balance)

- Can be split into two relations:

account-branch (<u>account-number</u>, branch-name)
account-balance (<u>account-number</u>, balance)

- The two representations are equivalent and in the maximum normal form (BCNF). But they could give different performance characteristics, depends on the information that usually retrieved at the same time. E.g. branch-name need not be fetched unless required.

## Schema Tuning: Denormalization (I)

- Can be done with tables or columns
- Assumes prior normalization
- Requires a thorough knowledge of how the data is being used
- Used if:
  - All or nearly all of the most frequent queries require access to the full set of joined data
  - A majority of applications perform table scans when joining tables
  - Computational complexity of derived columns requires temporary tables or excessively complex queries

## Schema Tuning: Denormalization (2)

- Denormalization can improve performance by:
  - Minimizing the need for joins
  - Reducing the number of foreign keys on tables
  - Reducing the number of indices, saving storage space, and reducing data modification time
  - Precomputing aggregate values
  - Reducing the number of tables (in some cases)
- Disadvantages:
  - It usually speeds retrieval but can slow data modification
  - It is always application-specific and must be re-evaluated if the application changes
  - It can increase the size of tables
  - In some instances, it simplifies coding; in others, it makes coding more complex

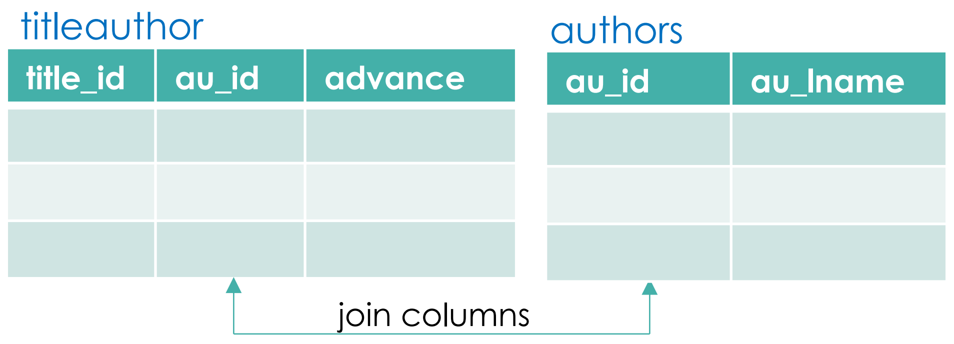
## Schema Tuning: Denormalization (3)

- Issues to examine when considering denormalization include:
  - What are the critical transactions, and what is the expected response time
  - How often are the transactions executed?
  - What tables or columns do the critical transactions use? How many rows do they access each time?
  - What is the mix of transaction types: select, insert, update, and delete
  - What is the usual sort order?
  - What are the concurrency expectations?
  - How big are the most frequently accessed tables?
  - Do any processes compute summaries?
  - Where is the data physically located?

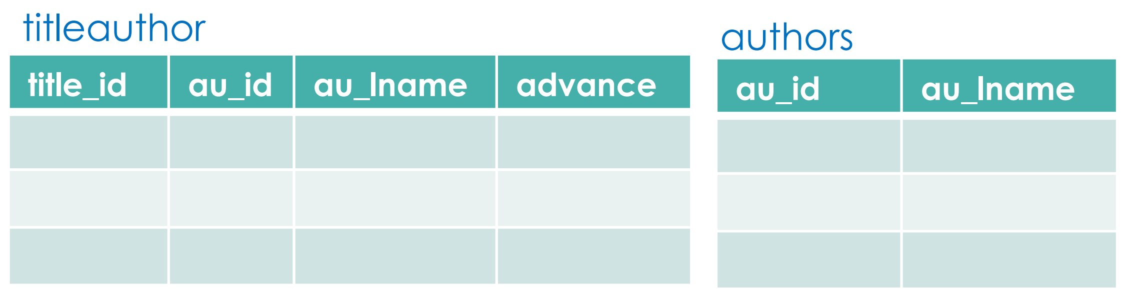
# *Schema Tuning: Denormalization*
## *Adding Redundant Columns*

- To eliminate frequent joins for many queries
- For example: if performing frequent joins on the ***titleauthor*** and ***authors*** tables to retrieve author's last name, you can add the ***au_lname*** column to ***titleauthor***
- The problems with this solution are that it:
  - Requires maintenance of new columns
    - Have to make changes to two tables, and possibly to many rows in one of the tables
  - Requires more disk space

```
select title, au_lname
from titleauthor ta, authors a
where ta.au_id = a.au_id
```



```
select title, au_lname from titleauthor
```



# Schema Tuning: Denormalization
## Adding Derived Columns

- Can eliminate some joins and reduce the time needed to produce aggregate values
- E.g.: The ***sum_adv*** column in the ***titles*** table
  - Eliminating both the join and the aggregate at runtime
  - Increases storage needs and requires maintenance of the derived column wherever changes are made to the titles table

```
select title, sum(advance)
from titleauthor ta, titles t
where ta.title_id = t.title_id
group by title
```

titleauthor

| title_id | advance |
|---|---|
| | |
| | |
| | |

titles

| title_id | title |
|---|---|
| | |
| | |
| | |

join columns

```
select title_id, sum_adv from titles
```

titleauthor

| title_id | advance |
|---|---|
| | |
| | |
| | |

titles

| title_id | title | sum_adv |
|---|---|---|
| | | |
| | | |
| | | |

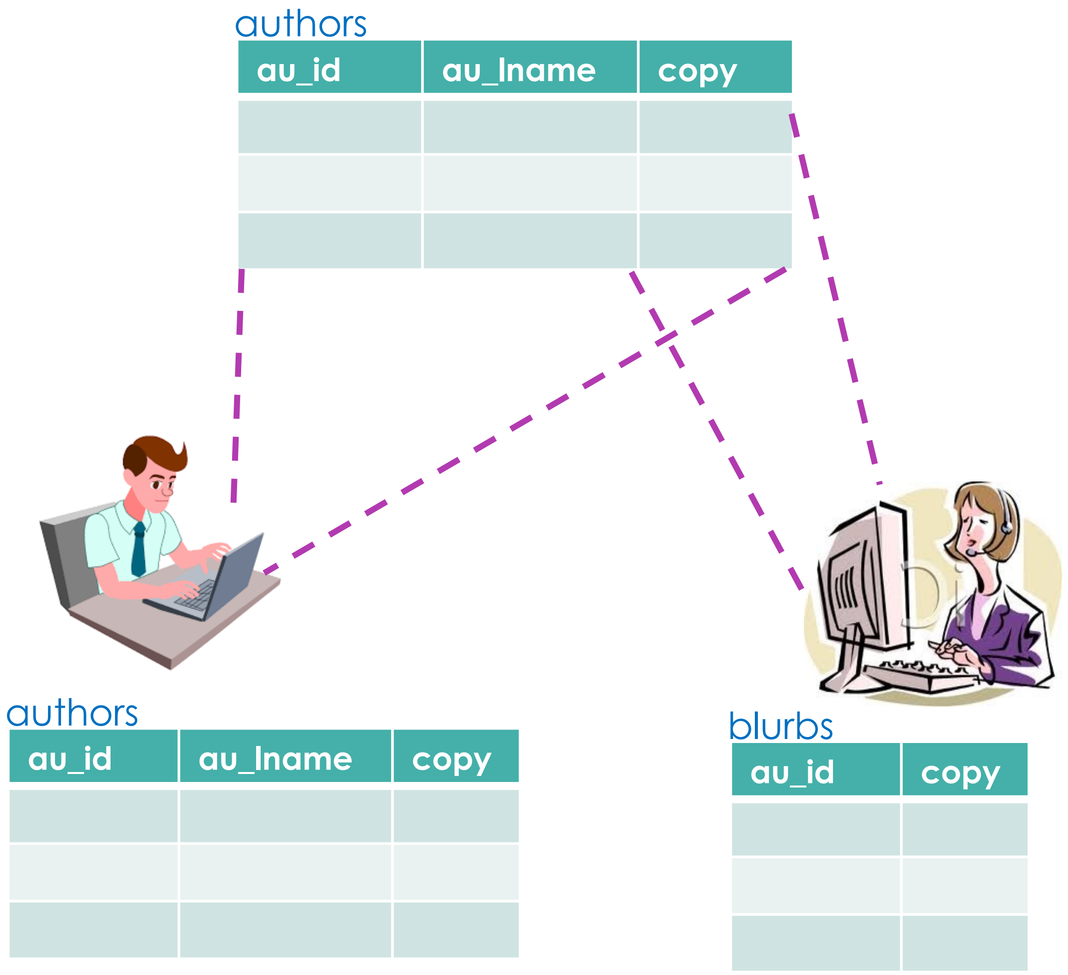
## Schema Tuning: Denormalization
### Collapsing Tables

- If most users need to see the full set of joined data from two tables, collapsing the two tables into one can improve performance by eliminating the join
- The data from two tables must be in a **one-to-one** relationship to collapse tables
- Eliminates the join, but loses the conceptual separation of the data
    - If some users still need access to just the pairs of data from the two tables, this access can be restored by using queries that select only the needed columns or by using views

# Schema Tuning: Denormalization

## Duplicating Tables

- If a group of users regularly needs only a subset of data, the critical table subset can be duplicated for that group
- Minimizes contention, but requires managing redundancy



## Schema Tuning: Managing Denormalized Data

- Need to ensure data integrity by using:
  - **Triggers**, which can update derived or duplicated data anytime the base data changes
  - **Application logic**, using transactions in each application that update denormalized data, to ensure that changes are atomic
    - Be very sure that the data integrity requirements are well documented and well known to all application developers and to those who must maintain applications
  - **Batch reconciliation**, run at appropriate intervals, to bring the denormalized data back into agreement
    - If 100 percent consistency is not required at all times
- From an integrity point of view, triggers provide the best solution, although they can be costly in terms of performance

# Tuning the Database Design: Materialized view (I)

- **Materialized views** can help speed up certain queries
  - Particularly aggregate queries
- Overheads
  - Space
  - Time for view maintenance
  - Immediate view maintenance: done as part of update transactions
    - time overhead paid by update transaction
  - Deferred view maintenance: done only when required
    - update transaction is not affected, but system time is spent on view maintenance
      - until updated, the view may be out-of-date
- Preferable to denormalized schema since view maintenance is systems responsibility, not programmers
  - Avoids inconsistencies caused by errors in update programs

## Tuning the Database Design: Materialized view (2)

- How to choose set of materialized views
  - Helping one transaction type by introducing a materialized view may hurt others
  - Choice of materialized views depends on costs
    - Users often have no idea of actual cost of operations
  - Overall, manual selection of materialized views is tedious
- Some database systems provide tools to help DBA choose views to materialize
  - "Materialized view selection wizards"

# Index Tuning

KNOWLEDGE & SOFTWARE ENGINEERING

# *Primary (Clustering) Indices*

- Primary (clustering) indices determines the order of physical records.
  - A **SELECT** statement with no **ORDER BY** clause will return data in the clustering index order.
  - 1 clustered index per table, 249 non-clustered indexes per table.
- Highly recommended for every table!
- Very useful for **columns** sorted on **GROUP BY** and **ORDER BY** clauses, as well as those filtered by WHERE clauses.

## Secondary (Non-Clustering) Indices

- Useful for retrieving a **single record** or a **range of records**.
- Maintained in a separate structure and maintained as changes are made to the base table.
- Tend to be **much smaller** than the base table, so they can locate the exact record(s) with much less I/O.
- Any time you **rebuild** the primary (clustering) index, you **also automatically rebuild** all secondary (non-clustering) index on the table.

# *Index Tuning*

- Create **appropriate indices to speed up** slow queries/updates.
- Speed up **slow updates by removing** excess indices (tradeoff between queries and updates).
- **Choose type of index** (B-tree/hash/Bitmap) appropriate for most frequent types of queries.
- Choose **which index** to be made **clustered**.
- **Index tuning wizards** look at past history of queries and updates (the workload) and **recommend** which indices would be best for the workload.

KNOWLEDGE & SOFTWARE ENGINEERING

## When to use each type of index? (I)

- Primary (clustered) indices are actual physically written records → Highly recommended for every table
  - Very useful for columns sorted in GROUP BY or ORDER BY as well as those filtered by WHERE clause
- Secondary (non-clustered) indices are useful for retrieving a single or a range of records
- B$^+$-tree
  - It takes approximately the same amount of time to access any given row in the index
  - The default and most common type of index used in DBMS
  - Used mainly in columns with values repeat in a relatively small number → high data sparsity

# When to use each type of index? (2)

- Hash index:
  - Good for simple and fast lookup operations based on equality conditions, e.g. LNAME = ‘Scott’; FNAME = ‘Shannon’
- Bitmap index:
  - Used mostly in data warehouse applications in tables with large number of rows in which a small number of column values repeat many times → low data sparsity

KNOWLEDGE & SOFTWARE ENGINEERING

## When should index be considered?

- **Unique index** are implicitly used in conjunction with a primary key for the primary key to work (dbms typically create index on primary-key)
- **Foreign keys** are also excellent candidates for an index because they are often used to join the parent table.
  - Most, if not all, columns used for table joins should be indexed.
- Columns that are frequently referenced in the **ORDER BY** and **GROUP BY** clauses should be considered for being indexed.
- **MAX** or **MIN** function is applied to index column

KNOWLEDGE & SOFTWARE ENGINEERING

## When should indexes be considered? (Cont.)

- Index should be created on columns with a **high number of unique values**, of columns when used as filter conditions in the **WHERE** clause **return a low percentage** of rows of data from a table
- The effective use of indexes requires a thorough knowledge of table relationships, query and transaction requirements, and the data itself.

## *Add an index when*

- Data within the column is **frequently accessed**, particularly if it is used to **filter** data either by **specific value** or by **range** of values
  - For example:
    - **WHERE emp_SSN = '123-56-7890'**
    - **WHERE emp_hire_date BETWEEN 'Jan 01, 2000' AND 'Nov 01, 2002'**
- Data within the column is **used to build joins**. Foreign keys are almost always good candidates for being indexed.
  - For example:

```
SELECT DISTINCT b.title
FROM book AS b
JOIN sales  AS s ON b.title_id = s.title_id
```

  - **title_ID** on the **sales** table should be indexed

KNOWLEDGE & SOFTWARE ENGINEERING

## *Index Tuning (cont.)*

When should indexes be avoided?

- Indexes should not be used **on small tables**.
- Indexes should not be used on columns that **return a high percentage of data** rows when used as a filter condition in a query's WHERE clause.
- Tables that have frequent, large batch update jobs run can be indexed.
  - However, the batch job's performance is slowed considerably by the index.
  - Might consider dropping the index before the batch job, and then recreating the index after the job has completed.
- Indexes should not be used on columns that contain a **high number of NULL** values.
- Columns that are **frequently manipulated** should not be indexed.
  - Maintenance on the index can become excessive.

# *Index Tuning (cont.)*

- **Rebuild** your index often
- Gather **statistics**
- Do not **overuse** indices
- Restrict to columns that **return a few records**
- Use **Bitmap index** when number of values is small e.g., sex: male, female, and filtering involving multi attributes
- Suppression of index

  select * from mytable where total + 3 = 20

KNOWLEDGE & SOFTWARE ENGINEERING

## How Many Indices?

- Adding **non-clustered indices** to a table can **greatly speed** **SELECT** statements.
- Every index has a certain amount of overhead.
  - The greater the number of indices, the more overhead with every INSERT, UPDATE, and DELETE statement.
- Must balance the needs of the application with the pros and cons of added indices
- **DSS** (Decision Support System) vs. **OLTP** (Online Transaction Processing)

## Index Tuning Wizard

- The Index Tuning Wizard allows you to select and create an optimal set of indices and statistics **without requiring an expert** understanding of the structure of the database, the workload, or the internals of db server
  - E.g. Microsoft® SQL Server™ 2000
- Returns best results using a realistic workload captured using SQL Profiler
- Recommendations returned by the wizard are SQL commands that can be run directly against the database in SQL Query Analyzer.