After logical design, we have a “good” logical schema that describes the data to be stored and related constraints.

The next step is physical database design, a process of producing a physical schema, i.e., a description of the implementation of the database on secondary storage using the target DBMS.

The main tasks include: choosing indexes, make clustering decisions, and refining 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

The problem is similar to query optimization, but this time for all queries in the workload.
Decisions to Make

- **Index selection**
  - 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? Hash/tree? Dense/sparse?

- **Should we make changes to the conceptual 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)
• A clustered index can be sparse: saves space and number of disk access
• Good for equality and range queries because the results will be in consecutive pages
• Good for index nested loop join if the join attribute is not a key of the inner relation
• If both relations have clustering index on join attributes, we can use merge join
• Non-clustered index has to be **dense**
  - best if it *covers* the query:
    - i.e., the query can be answered using *index-only scan*
  - very good for equality condition on a candidate key (each query retrieves a single record)
  - OK if each query retrieves only few records
  - **BAD** for queries that retrieve numerous records (e.g., range selection) because *sequential scan is much faster than numerous random accesses*
Attributes mentioned in a WHERE clause are candidates for index search keys

- Exact match condition suggests hash index (unless it retrieves a lot of records)
- Range query suggests tree index
  - Clustering is especially useful for range queries, although it can help on equality queries as well in the presence of duplicates

Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering
• Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - If range selections are involved, order of attributes should be carefully chosen to match the range ordering
  - Such indexes can sometimes enable index-only strategies for important queries
    • For index-only strategies, clustering is not important (you do not access the file anyway)!

• When considering a join condition:
  - Hash index on inner relation is very good for Index Nested Loops
  - Clustered tree index is good if the join column is not a key for the inner relation, and several inner relation tuples need to be retrieved
  - Clustered B⁺-Tree on join column(s) for both tables is good for Sort-Merge Join (eliminates the need for sorting)
• If we want to efficiently process the above query we can build the following indexes:
  - **Hash index on Dept.dname** supports equality selection on dname. Retrieves very few records (actually only one department is expected to have name="Toy").
  - Given this, index on Dept.dno is not needed (i.e., department becomes the outer relation in index nested loop).
  - **Hash index on Emp.dno** allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple. **Clustering B⁺-Tree** better because all employees of the same dept are stored together.

• What if **WHERE** included: “... AND E.age=40” ?
  - Could retrieve Emp tuples using **index on Emp.age**, then join with Dept tuples satisfying dname selection. Would need an index on **Hash index on Dept.dno**.
Emp should be the outer relation because of the selective conditions
  - Suggests that we build a hash index on Dept.dno.

What index should we build on Emp?
  - B⁺-Tree on Emp.sal, OR an index on E.age could be used. Only one of these is needed, and which is better depends upon the selectivity of the conditions
    • As a rule of thumb, equality selections are more selective than range selections, i.e., build index on age. Ideally the index should be clustered.
    • Can also build “composite index” on both attributes.
1. B+ tree index on E.age can be used to get qualifying tuples
   - Since the query retrieves a lot of records, index should be clustered in order to be useful

2. Since many tuples have $E.age \geq 25$, using $E.age$ index and sorting the retrieved tuples is costly
   - Clustered $E.dno$ index better!

3. Clustering is important when accessing inner tuples in indexed nested loop join
   - Should make index on $E.dno$ clustered
• To retrieve Emp records with \textit{age}=30 AND \textit{sal}=4000, an index on \textit{<age,sal>} would be better than an index on \textit{age} or an index on \textit{sal}

• If condition is: \textit{20<age<30 AND 3000<sal<5000}:  
  - Clustered tree index on \textit{<age,sal>} or \textit{<sal,age>} is best

• If condition is: \textit{age=30 AND 3000<sal<5000}:
  - Clustered \textit{<age,sal>} index better than \textit{<sal,age>} index!

• Composite indexes are larger, updated more often
• Can also use multidimensional indexes (e.g., R-trees, Grid files)
A number of queries can be answered by using only the index:

- \(<E.dno>\) dense index
- \(<E.dno,E.sal>\) Tree index!
- \(<E.age,E.sal>\)
  - or
- \(<E.sal,E.age>\) Tree index!

Clustering is not important since the file is not accessed at all.

```sql
SELECT  E.dno, COUNT(*)
FROM    Emp E
GROUP BY E.dno

SELECT  E.dno, MIN(E.sal)
FROM    Emp E
GROUP BY E.dno

SELECT AVG(E.sal)
FROM    Emp E
WHERE   E.age=25 AND E.sal BETWEEN 3000 AND 5000
```
The index selection problem is more complex in practice because it has to take into account:

- update and size constraints which restrict the number of indexes
- inter-dependencies between queries (e.g., the most efficient plans for two queries may require clustered indexes on different attributes)
Other Ways of Optimizing Performance

- **Denormalization**: introduce redundancy by adding attributes to tables
- **Materialized Views**: generate additional tables that contain the results of queries (increases overhead for updates and space requirements)
- **Horizontal Partitioning**: distributing the rows of a table into several separate files
  - Useful for situations where different users need access to different rows
- **Vertical Partitioning**: distribute the columns of a table into several separate files
  - Useful for situations where different users need access to different columns, or for aggregate queries on a single column (e.g., AVG(age))
  - The primary key must be repeated in each file
- **Combinations of Horizontal and Vertical Partitioning**