

Indexes

Lecture 11



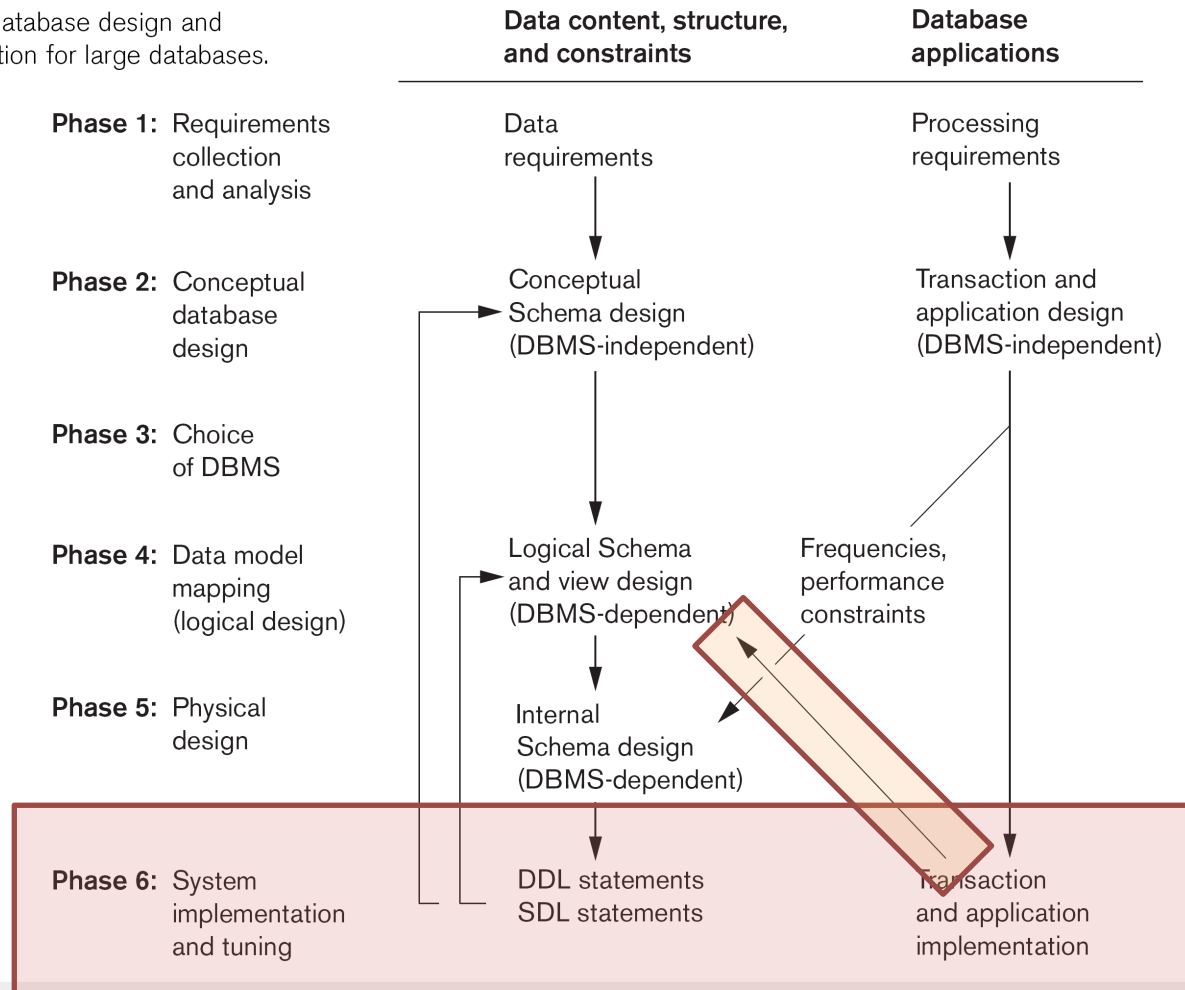
Outline

- Context
- Functionality
- Utility
- Tradeoffs and considerations
 - Selectivity
- Index types
- A reminder of syntax



Database Design and Implementation Process

Figure 10.1
Phases of database design and implementation for large databases.



What is an Index?

- Persistent data structure, stored in the database
- Primary mechanism to get improved query performance
- Many interesting issues (see Ch. 17-18); we will focus on usage, tradeoffs



Functionality

An index answers certain kinds of questions very efficiently (depends upon type)

- **Equality**: `fieldname=value`
- **Range/ordering**: `fieldname>value`
 - Only index that maintains ordering (e.g. tree-based)

Can be used for **WHERE** clause, as well as **JOIN** and **ORDER BY**



Comparison (1)

```
SELECT * FROM T
WHERE ...
```

- No indexes (indices)
Anything = full **table scan**
- Index on (A)
A = 'panda' (fast)
A > 'dog' (fast, if ordered index)
ORDER BY A (fast, if ordered)
- Index on (B)
B = 1 (fast)
B <= 5 (fast, if ordered)
ORDER BY B (fast, if ordered)
- Index on (A, B)
A = 'cat' (fast)
A = 'cat' AND B >= 3 (fast, if ordered)
A <= 'panda' ORDER BY B (fast, if ordered)
Anything not starting with A = full table scan
- Index on (C,A), (C,B), ... (i.e. start with C)
Anything not starting with C = full table scan

T	A	B	C
1	cat	1	...
2	dog	3	...
3	panda	7	...
4	cat	4	...
5	cat	5	...
6	panda	9	...
7	moose	10	...
8	dog	8	...
9	dog	10	...



Comparison (2)

T1	A	B	C	T2	X	Y	Z
1	cat	1	...	i	felidae	1	...
2	dog	3	...	ii	canidae	3	...
3	panda	7	...	iii	bear	7	...
4	cat	4	...	iv	felidae	4	...

T1 JOIN T2 ON T1.B=T2.Y

- No indexes: scan T1, scan T2 (n^2)
- Index on T1(B): scan T2, fast search in T1
- Index on T2(Y): scan T1, fast search in T2
- Index on T1(B), T2(Y): merge sort (if ordered)



Utility

Pro

- Can make the difference between full table scan and log/constant lookup

Con

- Extra space
 - Linear with # rows
- Extra time
 - Creation (moderate)
 - Maintenance (can offset savings)



Choosing the Index(es) to Create

- Table size
 - Many rows = larger cost to table scan
- Data distribution (selectivity)
 - Fewer distinct values = higher likelihood needing to touch many rows, independent of index usage
 - Index can lead to lots of IO/cache misses vs. sequential scan via clustered index
- Query vs. update load
 - Many updates = higher relative index maintenance cost
 - Analysis of frequent queries leads to choosing key attributes that get you the most bang for your buck



Selectivity

- **Cardinality**: # distinct values in a column

```
SELECT COUNT(DISTINCT col_name)
FROM table_name;
```

- **Selectivity**: $100\% * \text{cardinality} / \# \text{ rows}$
 - Compare for 10K rows...
 - Gender (M/F)
 - Country (195 + Taiwan)
 - Birthday (Jan. 1 -> Dec. 31)



General Advice

- Use narrow indexes (i.e. few columns); these are more efficient than compound indices
- Avoid a large number of indices on a table
- Avoid “overlapping” indices that contain shared columns (often a single index can service multiple queries)
- For indices that contain more than one column: given no other constraints, place the most selective column first
- Always define one clustered index on each table (typically equates to defining a PK)



Index Types

- Clustered vs. Non-clustered
- Covering (w.r.t. a query)
- Balanced Trees (B+-Trees)
- Hash Tables
- Other



Clustered vs. Non-clustered

- Clustered: affects physical order on disk
 - At most one per table (for some RDBMS, PK)
 - Fast when data accessed in order/reverse
- Non-clustered: induces logical ordering
 - Arbitrary number per table
 - Depending on the query/data, can lead to significant slowdown due to cache misses and frequent disk access



Covering

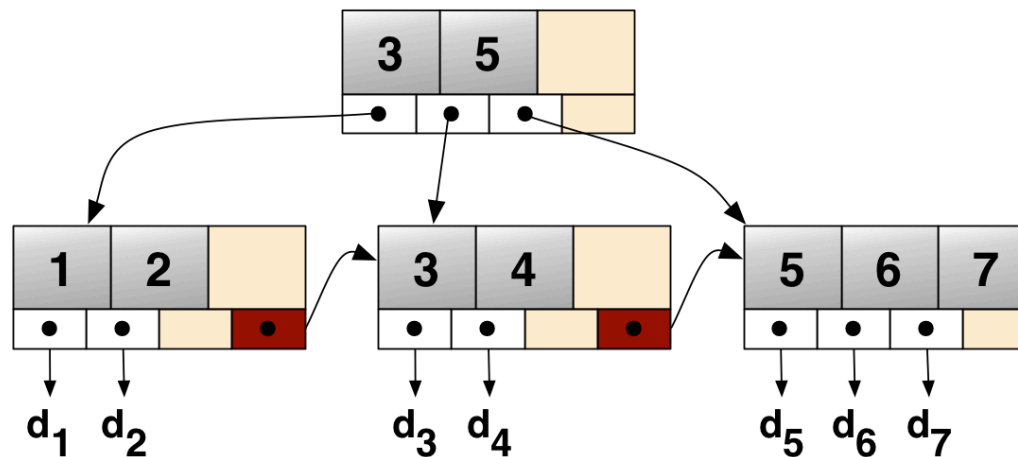
- Typically indexes help the DBMS *find* the row of interest
 - ID -> Name
 - Name->ID

ID	Name
1	Alice
2	Bob
3	Carol
4	Dan

- A covering index contains all the necessary data within the index itself (w.r.t. to query or queries)
 - More storage vs. IO savings
 - (ID, Name) or (Name, ID)



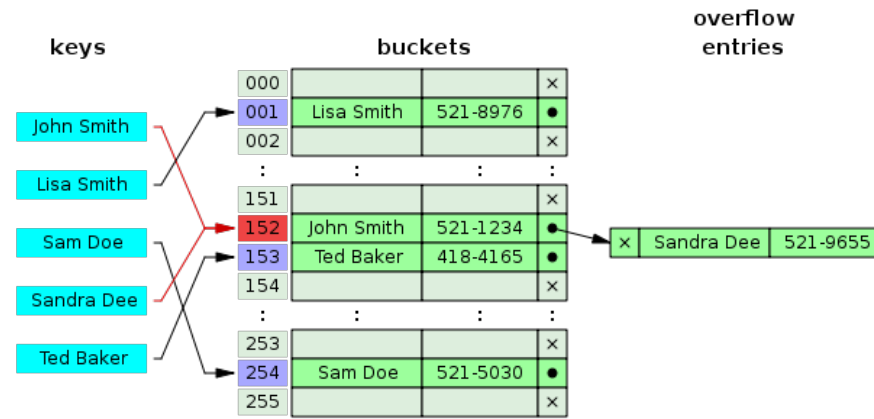
B+-Trees



- Balanced, constant out-degree (within range)
- Values (i.e. row pointer) only at leaves
 - Distinguishes from a B-tree
 - Linked list at leaves, in order
- Logarithmic traversal, constant at leaf
 - Top k levels usually kept in memory (e.g. 2-3)
- Typical default index for DBMS; also used in file systems, etc.



Hash Table



- “Constant” access time (under certain assumptions, amortized)
- No range queries



Other

- Bitmap
 - Useful for low-update systems (e.g. read-only) with low cardinality attributes (e.g. gender)
- Trie
 - Useful for sequence queries (e.g. bioinformatics)
- Spatial (e.g. Quadtree, R-tree)
 - Useful for queries about space (e.g. what stores are close to me? what planes are within 1 mile of each other?)
- Inverted
 - Useful for full-text search (e.g. search engines)



Creating Indexes

```
CREATE [UNIQUE] INDEX index_name  
ON table_name (c_name1, ...)  
[OPTIONS];
```

Notes

- Ordering of columns is **VERY** important
- Options often refer to the type of index being used (e.g. btree, hash) and other important flags

