Index Tuning

Index

An index is a data structure that supports efficient access to data

Condition on attribute value → index → Set of Records (search key) → Matching records
Performance Issues

- Type of Query
- Index Data Structure
- Organization of data on disk
- Index Overhead
- Data Distribution
- Covering

Types of Queries

1. Point Query
   SELECT balance
   FROM accounts
   WHERE number = 1023;

2. Multipoint Query
   SELECT balance
   FROM accounts
   WHERE branchnum = 100;

3. Range Query
   SELECT number
   FROM accounts
   WHERE balance > 10000;

4. Prefix Match Query
   SELECT *
   FROM employees
   WHERE name = 'Jensen'
   and firstname = 'Carl'
   and age < 30;
Types of Queries

5. **Extremal Query**
   
   ```sql
   SELECT *
   FROM accounts
   WHERE balance =
     max(select balance from accounts)
   ```

6. **Ordering Query**
   
   ```sql
   SELECT *
   FROM accounts
   ORDER BY balance;
   ```

7. **Grouping Query**
   
   ```sql
   SELECT branchnum, avg(balance)
   FROM accounts
   GROUP BY branchnum;
   ```

8. **Join Query**
   
   ```sql
   SELECT distinct branch.adresse
   FROM accounts, branch
   WHERE
     accounts.branchnum = branch.number
     and accounts.balance > 10000;
   ```

---

Index Tuning Knobs

- **Search key**
  - Size of key
- **Index data structure**
- **Clustered/Non-clustered/No index**
- **Covering**
Index Tuning Knobs

- Search key
  - Size of key
- Index data structure
  - Clustered/Non-clustered/No index
  - Covering

Search Keys

- A (search) key is a sequence of attributes.
  
  ```sql
  create index i1 on accounts(branchnum, balance);
  ```

- Types of keys
  - Sequential: the value of the key is monotonic with the insertion order (e.g., counter or timestamp)
  - Non sequential: the value of the key is unrelated to the insertion order (e.g., social security number)
Data Structures

- Most index data structures can be viewed as trees.
- In general, the root of this tree will always be in main memory, while the leaves will be located on disk.
  - The performance of a data structure depends on the number of nodes in the average path from the root to the leaf.
  - Data structure with high fan-out (maximum number of children of an internal node) are thus preferred.

B+-Tree

- A B+-Tree is a balanced tree whose leaves contain a sequence of key-pointer pairs.
**B+-Tree Performance**

- Key length influences fanout
  - Choose small key when creating an index
- Key compression
  - **Prefix compression** (Oracle 8, MySQL): only store that part of the key that is needed to distinguish it from its neighbors: Smi, Smo, Smy for Smith, Smoot, Smythe.
  - **Front compression** (Oracle 5): adjacent keys have their front portion factored out: Smi, (2)o, (2)y. There are problems with this approach:
    - Processor overhead for maintenance
    - Locking Smoot requires locking Smith too.

---

**Hash Index**

- A hash index stores key-value pairs based on a pseudo-randomizing function called a *hash function*.

![Hash Index Diagram]

The length of these chains impacts performance.
Hash Index Performance

- The best for answering point queries, provided there are no overflow chains
- Good for multipoint queries
  - Useless for range, prefix or extremal queries
- Must be reorganized (drop/add or use reorganize function) if there is a significant amount of overflow chaining
  - Avoiding overflow may require underutilize the hash space
- Size of hash structure is not related to the size of a key, because hash functions return keys to locations or page identifiers
  - Hash functions take longer to execute on a long key

Clustered / Non clustered index

- Clustered index (primary index)
  - A clustered index on attribute X co-locates records whose X values are near to one another.
- Non-clustered index (secondary index)
  - A non clustered index does not constrain table organization.
  - There might be several non-clustered indexes per table.
Dense / Sparse Index

- Sparse index
  - Pointers are associated to pages

- Dense index
  - Pointers are associated to records
  - Non clustered indexes are dense

Index Tuning Knobs

- Search key
  - Size of key

- Index data structure
  - Clustered/Non-clustered/No index

- Covering
Types of Queries

1. Point Query
   SELECT balance
   FROM accounts
   WHERE number = 1023;

2. Multipoint Query
   SELECT balance
   FROM accounts
   WHERE branchnum = 100;

3. Range Query
   SELECT number
   FROM accounts
   WHERE balance > 10000;

4. Prefix Match Query
   SELECT *
   FROM employees
   WHERE name = 'Jensen'
   and firstname = 'Carl'
   and age < 30;

5. Extremal Query
   SELECT *
   FROM accounts
   WHERE balance = max(select balance from accounts);

6. Ordering Query
   SELECT *
   FROM accounts
   ORDER BY balance;

7. Grouping Query
   SELECT branchnum,
   avg(balance)
   FROM accounts
   GROUP BY branchnum;

8. Join Query
   SELECT distinct
   branch.adresse
   FROM accounts, branch
   WHERE
   accounts.branchnum = branch.number
   and accounts.balance > 10000;
Benefits of a clustered index

1. A sparse clustered index stores fewer pointers than a dense index.
   - This might save up to one level in the B-tree index.
   - Nb pointers dense index = nb pointers sparse index * nb records per page
   - If records small compared to pages, there will be many records per page, so sparse has one level less than dense

2. A clustered index is good for multipoint queries
   - White pages in a paper telephone book

3. A clustered index based on a B-Tree supports range, prefix, extremal and ordering queries well

7. A clustered index (on attribute X) can reduce lock contention:
   - Retrieval of records or update operations using an equality, a prefix match or a range condition based on X will access and lock only a few consecutive pages of data

Evaluation of Clustered Index

- Multipoint query that returns 100 records out of 1000000.
- Cold buffer
- Clustered index is twice as fast as non-clustered index and orders of magnitude faster than a scan.
Inconvenient of a clustered index

- Benefits can diminish if there is a large number of overflow data pages
  - Accessing those pages will usually entail a disk seek
- Overflow pages can result from two kinds of updates:
  - Inserts may cause data pages to overflow
  - Record replacements that increase the size of a record (e.g., the replacement of a NULL values by a long string)

Evaluation of clustered indexes with insertions (1)

- Index is created with fillfactor = 100.
- Insertions cause page splits and extra I/O for each query
- Maintenance consists in dropping and recreating the index
- With maintenance performance is constant while performance degrades significantly if no maintenance is performed.
Evaluation of clustered indexes with insertions (2)

- Index is created with pctfree = 0
- Insertions cause records to be appended at the end of the table
- Each query thus traverses the index structure and scans the tail of the table.
- Performances degrade slowly when no maintenance is performed.

Evaluation of clustered indexes with insertions (3)

- In Oracle, clustered index are approximated by an index defined on a clustered table
- No automatic physical reorganization
- Index defined with pctfree = 0
- Overflow pages cause performance degradation
Redundant tables

- Because there is only one clustered index per table, it might be a good idea to replicate a table in order to use a clustered index on two different attributes
  - Yellow and white pages in a paper telephone book
  - Works well if low insertion/update rate

Index Tuning Knobs

- Search key
  - Size of key
- Index data structure
- Clustered/Non-clustered/No index
- Covering
Covering Index - definition

- A nonclustering index can eliminate the need to access the underlying table through covering (composite index)
- For the query:
  Select name
  from employee
  where department = "marketing"
- Good covering index would be on (department, name)
  - Index on (name, department) less useful.
  - Index on department alone moderately useful.

Inconveniences (of composite indexes):
- Tend to have a large key size
- Update to one of the attributes causes index to be modified

Covering Index - impact

- Covering index performs better than clustering index when first attributes of index are in the where clause and last attributes in the select.
- When attributes are not in order then performance is much worse.
Benefits of non-clustered indexes

1. A dense index can eliminate the need to access the underlying table through covering.
   - It might be worth creating several indexes to increase the likelihood that the optimizer can find a covering index.

2. A non-clustered index is good if each query retrieves significantly fewer records than there are pages in the table.
   - Point queries always useful
   - Multipoint queries useful if:
     \[ \text{number of distinct key values} > c \times \text{number of records per page} \]
     
     Where \( c \) is the number of pages retrieved in each prefetch.

Table Scan Can Sometimes Win

- IBM DB2 v7.1 on Windows 2000
- Range Query
- If a query retrieves 10% of the records or more, scanning is often better than using a non-clustering non-covering index.
Index on Small Tables

- Tuning manuals suggest to avoid indexes on small tables (containing fewer than 200 records)
- This number depends on the size of records compared with the size of the index key
  - If all data from a relation fits in one page (or in a single disk track and can be read into memory through a single physical read by prefetching)
    - an index adds at least an I/O
  - If each record fits in a page, then 200 records may require 200 disk accesses or more.
    - an index helps performance
  - If many inserts execute on a table with a small index, the index itself may become a concurrency control bottleneck
    - Lock conflicts near the root

Index on Small Tables and Updates

- Small table: 100 records
- Two concurrent processes perform updates (each process works for 10ms before it commits)
- No index: the table is scanned for each update. No concurrent updates.
- A clustered index allows to take advantage of row locking.
Table organization and index selection: basic rules (1)

1. Use a hash index for point queries only. Use a B-tree if multipoint queries or range queries are used.
2. Use clustering
   - if your queries need all or most of the fields of each record returned, but the records are too large for a composite index on all fields
   - if multipoint or range queries are asked
3. Use a dense index to cover critical queries
4. Don’t use an index if the time lost when inserting and updating overwhelms the time saved when querying.

Table organization and index selection: basic rules (2)

- Use key compression
  - If you are using a B-tree
  - Compressing the key will reduce the number of levels in the tree
  - The system is disk-bound but not CPU-bound
  - Updates are relatively rare
## Index Implementations in some major DBMS

<table>
<thead>
<tr>
<th>DB</th>
<th>Index Structure</th>
<th>Clustered Indexes</th>
<th>Index Maintenance</th>
<th>Clustering &amp; Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server</td>
<td>B+-Tree data structure</td>
<td>Sparse</td>
<td>Updates/Insertions/Deletes</td>
<td>Updates/Insertions/Deletes performed</td>
</tr>
<tr>
<td>DB2</td>
<td>B+-Tree data structure, spatial extender for R-tree</td>
<td>Dense</td>
<td></td>
<td>Explicit command for index reorganization</td>
</tr>
<tr>
<td>Oracle</td>
<td>B+-Tree, hash, bitmap, spatial extender for R-Tree</td>
<td>No clustered index until 10g</td>
<td></td>
<td>Index organized table (unique/clustered) Clusters used when creating tables.</td>
</tr>
<tr>
<td>MySQL</td>
<td>B+-Tree, R-Tree (geometry and pairs of integers)</td>
<td></td>
<td></td>
<td>Updates/Insertions/Deletes performed</td>
</tr>
</tbody>
</table>