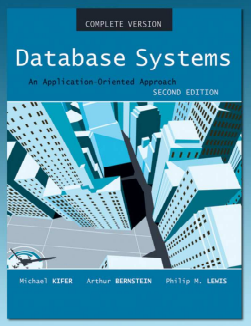


Chapter 12

Database Tuning

1

Improving the Performance of an Application

- Performance is generally measured by:
  - **Response time** – average time to wait for a response to a particular query
  - **Throughput** – volume of work completed in a fixed amount of time (often measured as transactions per second)

2

Tuning

- Measures taken to improve performance.
  - Application level:
    - Query: schema redesign, use of indexes, client vs. server processing (stored procedures)
    - Transaction: isolation level, code design
  - System level:
    - Cache issues: cache size, binding, I/O size
    - Distribution of data across devices
    - Log management
  - Hardware level:
    - Number of cpu's
    - Disk configuration (many small disks vs. a few large ones)
    - Backup (mirrored disks vs logs)
  - Distribution:
    - Replication
    - Distributing data and processing

3

Schema Redesign: Denormalization

- Normalization reduces redundancy and avoids anomalies
- Normalization can improve performance
  - Less redundancy => more rows/page => less I/O
  - Decomposition => more tables => more clustered indexes => smaller indexes

4

Normalization

- Normalization can decrease performance.
- **Example**: Transcript(*StudId*, *CrsCode*, *Semester*, *Grade*)
  - Functional dependency: *StudId* → *Name*
  - Key of Transcript = (*StudId*, *CrsCode*, *Semester*)
  - If *Name* were an attribute of Transcript it would not be in BCNF or 3NF, *but* …
  - a join required to list names of students with A in CS305

```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
                      AND T.Grade = 'A'
```

  - … and join is expensive

5

Denormalize

- Add attribute Name to Transcript

```
SELECT T.Name
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Grade = 'A'
```

- Join avoided, but added redundancy:
  - Slows data modification (update to redundant attribute has to be performed in two tables)
  - Increases size of table
  - Introduces the possibility of inconsistent data

6

Schema Redesign: Partitioning of Tables

- A table might be a performance bottleneck
  - If it is heavily used, causing locking contention
  - If it's index is deep (table has many rows or search key is wide), increasing I/O
  - If rows are wide, increasing I/O
- Table partitioning might be a solution to this problem

7

Horizontal Partitioning

- If accesses are confined to disjoint subsets of rows, partition table into smaller tables containing the subsets
  - Geographically (e.g., by state), organizationally (e.g., by department), active/inactive (e.g., current students vs. grads)
- Advantages:
  - Spreads users out and reduces contention (particularly if tables can be spread over different devices)
  - Indexes have fewer levels (but only marginally)
  - Rows in a typical result set are concentrated in fewer pages
- Disadvantages:
  - Added complexity
  - Difficult to handle queries over all tables

8

Vertical Partitioning

- Split columns into two subsets and replicate key
- Useful when table has many columns and
  - it is possible to distinguish between frequently and infrequently accessed columns
  - different queries use different subsets of columns
- **Example**: Employee table
  - Columns related to compensation (tax, benefits, salary) split from columns related to job (department, projects, skills).
  - Since SSN is included in each set, it is possible to retrieve all information about each employee (decomposition is lossless), although a join is required.

9

Extents and Storage Structures

- **Extent**: a group of contiguous blocks on mass store that serves as an allocation unit for a table or index
  - Allocating an extent for a table keeps its pages together and reduces latency: if another page is needed
    - a seek is avoided with high probablility
    - alternatively, the entire extent can be retrieved at a cost not much greater than the cost of retrieving a single page
- **Storage structure**:
  - Heap: unsorted rows in a file (table without a clustered index)
  - Integrated table and index (sparse, clustered B^+ tree or hash) in a file
  - Sorted rows in a file (dense, clustered index stored separately)

10

Heap

- Created when no primary key or unique constraint is declared in CREATE TABLE
- If there is no useful index then SELECT, UPDATE, and DELETE scan entire table
  - Excessive I/O
  - Contention since a transaction must lock entire table
- Rows are INSERTed at end
  - Contention since all transactions must lock last page (exclusively)

11

Indexes

- Some advantages of using an index:
  - Avoid table scan
    - In some cases, avoid accessing the table entirely
  - Enforce uniqueness
  - Randomize inserts
  - Support joins
- Created automatically to enforce primary key or unique constraint

12

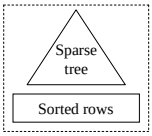
Clustered Index

- Only one clustered index can be created for a table since clustering specifies how the rows are to be stored.
- Generally created automatically based on PRIMARY KEY constraint.

13

Integrated Storage Structure: B+ Tree

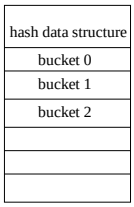
- A clustered index implemented as a sparse tree over sorted rows
  - avoids table scan for many SQL statements
  - supports range as well as point queries
  - but, data page (in addition to index page) splitting necessary to keep rows sorted



14

Integrated Storage Structure: Hash

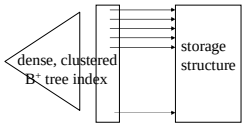
- A set of buckets with an associated hash is also possible
  - Does not support range queries



15

Clustered Index over Sorted File

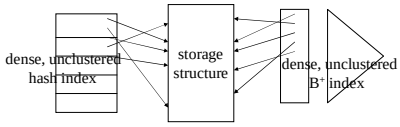
- Dense, clustered B+ tree index, stored separately, refers to (mostly) sorted file
  - Avoids many data page splits: rows do not have to be in exact sorted order since index is dense
    - If row doesn't fit in page, store it in another page in same extent
  - Supports same searches as integrated storage structure, but efficiency drops if table is dynamic



16

Unclustered Index

- Dense index (hash or B+ tree), stored in separate file
  - Created automatically when UNIQUE constraint declared
  - An arbitrary number of unclustered indexes possible
  - Same searches as clustered index, but less efficient:
    - Index entries and rows ordered differently
    - Additional level in tree
  - Supports index covering
  - Adds overhead when table is modified



17

Explicit Index Creation

- Indices can also be created explicitly using (proprietary) commands of the DBMS
  - CREATE CLUSTERED INDEX *index_name* ON *table_name* (*search_key_attribute_list*)
    - Causes storage structure to be reorganized
  - CREATE UNCLUSTERED INDEX *index_name* ON *table_name* (*search_key_attribute_list*)

18

Schema for Examples

- Student (Id, Name, Address, …)
  - Primary key: Id
- Professor (Id, Name, DeptId, Salary, …)
  - Primary key: Id
- Department (Id, Name, …)
  - Primary key: Id
- Transcript (StudId, CrsCode, Semester, Grade)
  - Primary key: (StudId, CrsCode, Semester)

19

Index Covering

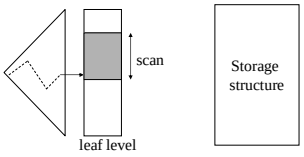
- If all attributes (in all clauses) of a query are included in the search key of a dense (clustered or unclustered) B^+ tree index, then the result set can be computed from the index alone.

20

Index Covering

– **Matching case:** attributes used in WHERE clause include a *prefix* of search key
  - Descend from index root and scan segment of leaf level
  - **Ex.** - Dense index on (DeptId, Name) covers the query:

```
SELECT P.Name
FROM Professor P
WHERE P.DeptId = 'CS'
```

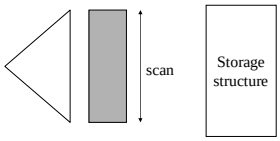


21

Index Covering

– **Non-matching case**: attributes in WHERE clause do not include a prefix of search key
  - Scan entire leaf level
  - **Ex.** - Dense index on (Id, Name, Address) covers the query:

```
SELECT S. Id, S.Name
FROM Student S
WHERE S.Address = '1 Lake St'
```

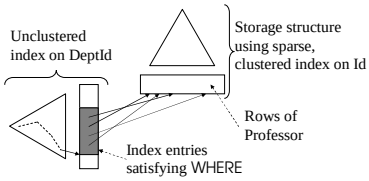


22

Choosing an Index – Example 1

```
SELECT P.Id, P.Name
FROM Professor P
WHERE P.DeptId = 'CS'
```

- Choose an index on DeptId
  – If a clustered index already exists on Id (since it is primary key), then index on DeptId is unclustered

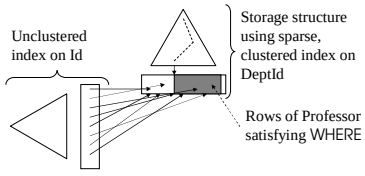


23

Choosing an Index – Example 1

```
SELECT P.Id, P.Name
FROM Professor P
WHERE P.DeptId = 'CS'
```

- But an unclustered index is a bad idea if result set is large
  – Choose an unclustered index for primary key (Id) and a clustered index on DeptId



24

Choosing an Index – Example 2

```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
      (Join condition)    (Select condition)
```

- If no useable index available, DBMS can
  - use block-nested loops join based on join condition
  - pipe to a selection using select condition
  - Not good – most rows satisfying join condition fail select condition

Choosing an Index – Example 2

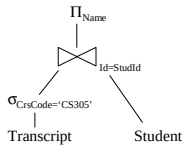
```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
      (Join condition)    (Select condition)
```

- Alternate plan:
  - Choose clustered index on Transcript with search key (CrsCode)
    - A clustered index with search key (CrsCode, Semester, StudId) might already exist since it corresponds to primary key of Transcript
  - Choose index on Student with search key (Id)
    - An index with search key (Id) already exists since it is primary key
    - Index can be B^+ tree or hash, clustered or unclustered

Choosing an Index – Example 2

```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
```

- DBMS can then use an index-nested loops join:
  - Retrieve all Transcript rows satisfying selection condition (they are stored contiguously since index on CrsCode is clustered)
  - For each such row use index on Student to retrieve *unique* row satisfying join condition (index-nesting particularly effective in this case)



Choosing an Index – Example 3

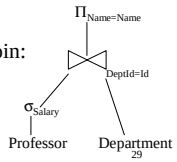
```
SELECT P.Name, D.Name
FROM Professor P, Department D
WHERE P.Salary BETWEEN 60000 AND 70000
      AND P.DeptId = D.Id
```

- Choose index on Professor with search key (Salary)
  - Should be B^+ tree to support range
  - Should be clustered since many matches

Choosing an Index – Example 3

```
SELECT P.Name, D.Name
FROM Professor P, Department D
WHERE P.Salary BETWEEN 60000 AND 70000
      AND P.DeptId = D.Id
```

- Choose index on Department with search key (Id)
  - Hash or B+ tree since a unique department corresponds to a row of Professor
  - Unclustered is sufficient (same reason)
    - Don't waste a clustered index
- DBMS can use index-nested loops join:
  - Retrieve Professor rows using clustered index on Salary
  - For each such row get matching row of Professor using index on Id



Choosing an Index – Example 4

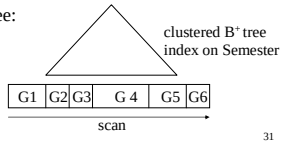
```
SELECT T.Semester, COUNT(*)
FROM Transcript T
WHERE T.Grade < 'A'
GROUP BY T.Semester
```

- Choose index on Grade
  - Use index to retrieve rows satisfying WHERE
    - B^+ tree since a range is specified
    - clustered since many rows satisfy WHERE
  - Sort result on Semester and count size of each group
- Not a good plan since WHERE condition not selective and a large intermediate table must be sorted
  - Might be acceptable if conditions were T.Grade < 'C'

Choosing an Index – Example 4

```
SELECT T.Semester, COUNT(*)
FROM Transcript T
WHERE T.Grade < 'A'
GROUP BY T.Semester
```

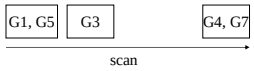
- Choose index on Semester
  - Use a clustered index so that rows will be grouped
  - Scan Transcript, counting qualifying rows in each group
  - Index can be B+ tree:



Choosing an Index – Example 4

```
SELECT T.Semester, COUNT(*)
FROM Transcript T
WHERE T.Grade < 'A'
GROUP BY T.Semester
```

- Index on Semester can be a hash:
  - A hash is acceptable since query *does not* use a range on Semester
  - Hash stores all rows of a particular group in one bucket; if a bucket fits in memory it is easy to count the qualifying rows in each group it contains
  - Scan buckets



- Good plan since WHERE is not selective; if it were, scan might be too costly

Choosing an Index – Example 5

(1)
```
SELECT T.CrsCode
FROM Transcript T
WHERE T.StudId = :studid
```

(2)
```
SELECT T.StudId
FROM Transcript T
WHERE T.CrsCode = :code
 AND T.Semester = :sem
```

- Two frequently asked queries
  - Sol'n 1: clustered index on StudId for (1), unclustered index on (CrsCode, Semester) for (2)
    - Problem: both result sets are moderate size, using an unclustered index for (2) results in excessive overhead
  - Sol'n 2: clustered index on (CrsCode, Semester) for (2), unclustered index on (StudId, CrsCode) for (1)
    - (1) uses index covering (matching case)

Choosing a Clustered Index

- Choose clustered index to support:
  - Point queries with large result sets
    - A clustered index on Transcript with search key (CrsCode, Semester, StudId) supports queries requesting the Id's of all students in a particular class.
  - Range queries
    - A clustered index on Transcript with search key (Grade) supports queries requesting the Id's of all students with grades between B+ and A. (In this case the primary key constraint must be enforced with an unclustered index.)
  - ORDER BY clauses
    - Sequence of attributes in ORDER BY is a prefix of search key
  - Index-nested (get all rows satisfying a particular search key value) and sort-merge (avoid sort) joins.

Choosing a Clustered Index

- Do not choose a clustered index
  - If it is not needed for one of the above
  - If the search key attribute is frequently updated
    - Since it will be necessary to move rows frequently
      - Hash – from bucket to bucket
      - B+ tree – from one leaf page to another (contention results if index pages near the root need to be modificd - use an ISAM index in this case)
  - B+ tree: if search key is primary key and is monotonically increasing with each insert (since all inserts go into last leaf page, causing contention)
    - E.g., invoice number, date, time

Choosing an Unclustered Index

- Choose unclustered index to support
  - Queries with small result sets
  - Index-nested loop joins (when the weight of the join attribute is small)
  - Index covering
  - Point queries with small result sets

Miscellaneous Hints

- If an attribute is unique, declare it so (query optimizer can use it in query planning)
  - Only one row can match a search or join attribute
- Adjust fill factor
  - To 100% if table is read-only
  - To a smaller value if table is dynamic
- Keep search keys small to flatten index
- Add unclustered indexes only when necessary

37

Tuning SQL

- Hints:
  - Avoid sorts
    - Sort-merge join
    - Use of DISTINCT, UNION, EXCEPT, ORDER BY, GROUP BY cause sorts. Avoid their use if possible.
  - Minimize communication
    - Don't use cursors (since communication may be required for each row retrieved)
    - Use stored procedures if only aggregrate information is required by the application

38

Tuning SQL

- Hints (con't):
  - Beware of views since they may cause unnecessary joins
  - Consider restructuring a query – different formulations will have different costs depending on state of tables and indexes available.

39

Influencing the Query Optimizer

- Statistics: Used by opimizers to estimate cost of a query plan (based on size of result sets)
  - Table: number of rows, number of distinct values of an attribute, max and min attribute values
  - Index: depth, number of leaf pages, number of distinct search key values
  - Histograms of attribute values
    - Example: use unclustered index if histogram shows that number of rows with attribute value specified in query is small, else use scan
  - Must be periodically updated if table dynamic

40

Influencing the Query Optimizer

- Hints: suggestions inserted into an SQL statement for consideration by optimizer
  - Join order
  - Join method
  - Index to use
  - ……

41

System Level Tuning – The Cache

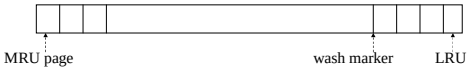
- Store recently referenced pages in main memory since probability is high that they will be referenced again.
- Essential to achieve realistic performance goals: a hit rate of at least 90% is sought.
- Used for data pages (data cache) and query plan pages (procedure cache)

42

Clean/Dirty Pages

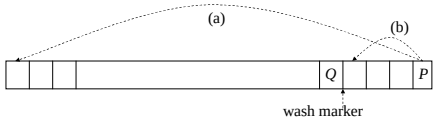
- Cache pages that have been updated are marked *dirty*; others are *clean*
- Cache ultimately fills
  - Clean pages can simply be overwritten
  - Dirty pages must be written to database before page frame can be reused
    - It is more efficient to overwrite a clean page

LRU Page Replacement Policy



- LRU page is overwritten when new page is brought in
  - New page becomes MRU page
- Reference to a page causes it to move to MRU position
- Dirty page must be written before reaching LRU end to avoid delays
  - a write is initiated when a page reaches the wash marker
  - wash marker must be carefully set so that:
    - probability is high that page is clean by the time it reaches LRU end
    - probability is high that page is *not* referenced after passing marker (else it might be written several times before it is overwritten)

MRU Page Replacement Policy



- LRU strategy (a): inefficient for a new page, *P*, that is unlikely to be referenced a second time (table scan, outer loop of nested join) since it unecessarily forces *Q* into the write area
- MRU strategy (b): Keep *P* in write area (it will not be overwritten until it reaches LRU end)
- Information about how a page is being used is contained in query plan

Fetching Pages Into The Cache

- **I/O Size**: Retrieve multiple pages (in an extent) of a table with a single I/O (and latency)
  - cache might support several different transfer sizes (e.g., 1, 2 4, … pages from same extent)
    - Multiple pages retrieved at cost of a single latency
  - Useful for table that is heavily referenced
- **Prefetch**: Retrieve a page before it has been requested
  - Useful for a table scan

Partitioning the Cache

- **Named caches and binding**: Default cache can be divided into several (named) caches of specified sizes
  - Table/index can be bound to a particular cache
  - A page in one cache cannot be overwritten by a page of an object bound to a different cache
  - Useful for queries that have stringent response times
  - By binding objects to caches application programmer can exert some control over page replacement policy

System Level Tuning: the Log

- Transaction log is an example of a heap storage structure: records appended at end.
- Place log on a separate device:
  - Avoids contention with access to database
  - Avoids seeks since head is always on correct track