# Database Management and Performance Tuning

## Index Tuning

Pei Li

University of Zurich
Institute of Informatics

Unit 4

Acknowledgements: The slides are provided by Nikolaus Augsten
and adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

# Outline

# Query Types

- Different indexes are good for different query types.
- We identify categories of queries with different index requirements.

# Query Types

- Point query: returns at most one record

```
SELECT name
FROM Employee
WHERE ID = 8478
```

- Multipoint query: returns multiple records based on equality condition

```
SELECT name
FROM Employee
WHERE department = 'IT'
```

- Range query on $X$ returns records with values in interval of $X$

```
SELECT name
FROM Employee
WHERE salary >= 155000
```

# Query Types

- Prefix match query: given an ordered sequence of attributes, the query specifies a condition on a prefix of the attribute sequence
- Example: attribute sequence: lastname, firstname, city
  - The following are prefix match queries:
    - `lastname='Gates'`
    - `lastname='Gates' AND firstname='George'`
    - `lastname='Gates' AND firstname like 'Ge%'`
    - `lastname='Gates' AND firstname='George' AND city='San Diego'`
  - The following are not prefix match queries:
    - `firstname='George'`
    - `lastname LIKE '%ates'`

# Query Types

- Extremal query: returns records with max or min values on some attributes

```
SELECT name
FROM Employee
WHERE salary = MAX(SELECT salary FROM Employee)
```

- Ordering query: orders records by some attribute value

```
SELECT *
FROM Employee
ORDER BY salary
```

- Grouping query: partition records into groups;
  usually a function is applied on each partition

```
SELECT dept, AVG(salary)
FROM Employee
GROUP BY dept
```

# Query Types

- Join queries: link two or more tables
- Equality join:

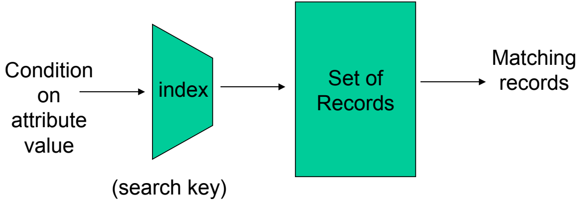
```
SELECT Employee.ssnum
FROM Employee, Student
WHERE Employee.ssnum = Student.ssnum
```

- Join with non-equality condition:

```
SELECT e1.ssnum
FROM Employee e1, Employee e2
WHERE e1.manager = e2.ssnum
AND e1.salary > e2.salary
```

# What is an Index?

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




- Index tuning essential to performance!
- Improper index selection can lead to:
  - indexes that are maintained but never used
  - files that are scanned in order to return a single record
  - multitable joins that run for hours or days

# Key of an Index

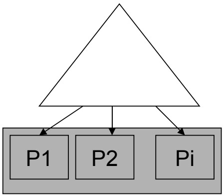
- Search key or simply "key" of an index:
  - single attribute or sequence of attributes
  - values on key attributes used to access records in table
- Sequential Key:
  - value is monotonic with insertion order
  - examples: time stamp, counter
- Non-sequential Key:
  - value unrelated to insertion order
  - examples: social security number, last name
- Note: index key different from key in normalization theory
  - normalization theory: key attributes have unique values
  - index key: not necessarily unique

# Index Characteristics

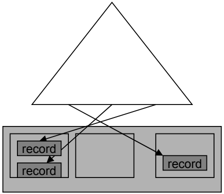
- Indexes can often be viewed as trees ($B^+$-tree, hash)
  - some nodes are in main memory (e.g., root)
  - nodes deeper down in tree are less likely to be in main memory
- Number of levels: number of nodes in root-leaf path
  - a node is typically a disk block
  - one block read required per level
  - reading a block costs several milliseconds (involves disk seek)
- Fanout: number of children a node can have
  - large fanout means few levels
- Overflow strategy: insert into a full node $n$
  - $B^+$-tree: split $n$ into $n$ and $n'$, both at same distance from root
  - overflow chaining: $n$ stores pointer to new node $n'$

# Sparse vs. Dense

- Sparse index: pointers to disk pages
  - at most one pointer per disk page
  - usually much less pointers than records




- Dense index: pointers to individual records
  - one key per record
  - usually more keys than sparse index
  - optimization: store repeating keys only once, followed by pointers



# Sparse vs. Dense

- Number of pointers:

$$\text{ptrs in dense index} = \text{records per page} \times \text{ptrs in sparse index}$$

- Pro sparse: less pointers
  - typically record size is smaller than page size
  - less pointers result in less levels (and disk accesses)
  - uses less space
- Pro dense: index may "cover" query

# Covering Index

- Covering index:
  - answers read query within index structure
  - fast, since data is not accessed
- Example 1: dense index on lastname

  `SELECT COUNT(lastname) WHERE lastname='Smith'`
- Example 2: dense index on `A, B, C` (in that order)
  - covered query:

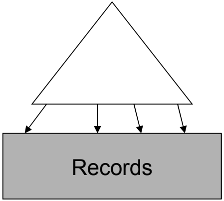
    ```
    SELECT B, C
    FROM R
    WHERE A = 5
    ```
  - covered query, but not prefix:

    ```
    SELECT A, C
    FROM R
    WHERE B = 5
    ```
  - non-covered query: `D` requires data access

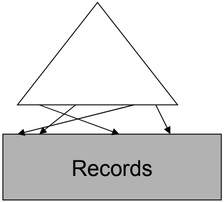
    ```
    SELECT B, D
    FROM R
    WHERE A = 5
    ```

# Clustering vs. Non-Clustering

- Clustering index on attribute $X$
  (also *primary index*)
  - records are grouped by attribute $X$ on disk
  - $B^+$-tree: records sorted by attribute $X$
  - only one clustering index per table
  - dense or sparse




- Non-clustering index on attribute $X$
  (also *secondary index*)
  - no constraint on table organization
  - more than one index per table
  - always dense



# Clustering Indexes

- Can be sparse:
  - fewer pointers than non-clustering index (always dense!)
  - if record is small, save one disk access per record access
- Good for multi-point queries:
  - equality access on non-unique attribute
  - all result records are on consecutive pages
  - example: look up last name in phone book
- Good for range, prefix, ordering queries:
  - works if clustering index is implemented as $B^+$-tree
  - prefix example: look up all last names starting with 'St' in phone book
  - result records are on consecutive pages
- Good for equality join:
  - fast also for join on non-key attributes
  - index on one table: indexed nested-loop
  - index on both tables: merge-join
- Overflow pages reduce efficiency:
  - if disk page is full, overflowing records go to overflow pages
  - overflow pages require additional disk accesses

# Equality Join with Clustering Index

- Example query:

```
SELECT Employee.ssnum, Student.course
FROM Employee, Student
WHERE Employee.firstname = Student.firstname
```

- Index on Emplyee.firstname: use index nested loop join
  - for each student look up employees with same first name
  - all matching employees are on consecutive pages
- Index on both firstname attributes: use merge join
  - read both tables in sorted order and merge ($B^+$-tree)
  - each page read exactly once
  - works also for hash indexes with same hash function

# Clustering Index and Overflow Pages

- Why overflow pages?
  - clustering index stores records on consecutive disk pages
  - insertion between two consecutive pages not possible
  - if disk page is full, overflowing records go to overflow pages
- Additional disk access for overflow page: reduced speed
- Overflow pages can result from:
  - inserts
  - updates that change key value
  - updates that increase record size (e.g., replace NULL by string)
- Reorganize index:
  - invoke special tool
  - or simply drop and re-create index

# Overflow Strategies

- Tune free space in disk pages:
  - Oracle, DB2: `pctfree` (0 is full), SQLServer: `fillfactor` (100 is full)
  - free space in page is used for new or growing records
  - little free space: space efficient, reads are faster
  - much free space: reduced risk of overflows
- Overflow strategies:
  - split: split full page into two half-full pages and link new page e.g., $A \rightarrow B \rightarrow C$, splitting $B$ results in $A \rightarrow B' \rightarrow B'' \rightarrow C$ (SQLServer)
  - chaining: full page has pointer to overflow page (Oracle)
  - append: overflowing records of all pages are appended at the end of the table (DB2)

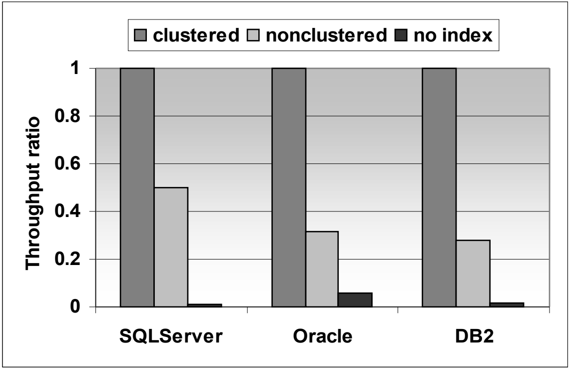
# Non-Clustering Index

- Always useful for point queries.
- Particularly good if index covers query.
- Critical tables: covering index on all relevant attribute combinations
- Multi-point query (not covered): only if not too selective
  - $nR$: number of records returned by query
  - $nP$: number of disk pages in table
  - the $nR$ records are uniformly distributed over all pages
  - thus query will read $\min(nR, nP)$ disk pages
- Index may slow down highly selective multi-point query:
  - scan is by factor 2–10 faster than accessing all pages with index
  - thus $nR$ should be significantly smaller than $nP$

# Non-Clustering Index and Multi-point Queries – Example

- Example 1:
  - records size: $50B$
  - page size: $4kB$
  - attribute $A$ takes 20 different values (evenly distributed among records)
  - does non-clustering index on $A$ help?
- Evaluation:
  - $nR = m/20$ ($m$ is the total number of records)
  - $nP = m/80$ (80 records per page)
  - $m/20 > m/80$ thus index does not help
- Example 2: as above, but record size is $2kB$
- Evaluation:
  - $nR = m/20$ ($m$ is the total number of records)
  - $nP = m/2$ (2 records per page)
  - $m/20 << m/2$ thus index might be useful

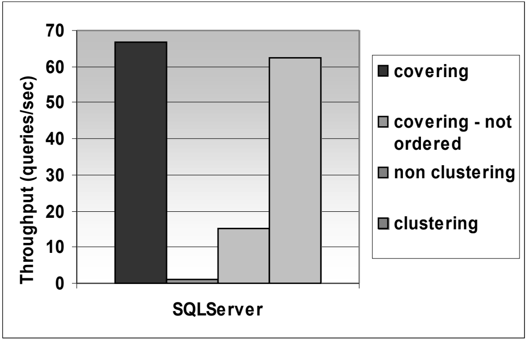
# Clustering vs. Non-Clustering Index




- multi-point query with selectivity 100/1M records (0.01%)
- clustering index much faster than non-clustering index
- full table scan (no index) orders of magnitude slower than index

DB2 UDB V7.1, Oracle 8.1, SQL Server 7 on Windows 2000

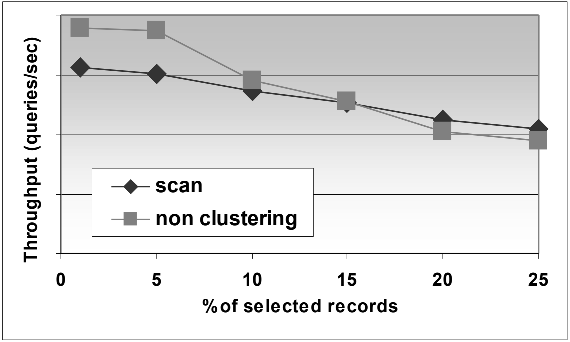
# Covering vs. Non-Covering Index




- prefix match query on sequence of attributes
- covering: index covers query, query condition on prefix
- covering, not ordered: index covers query, but condition not prefix
- non-clustering: non-covering index, query condition on prefix
- clustering: sparse index, query condition on prefix

SQL Server 7 on Windows 2000

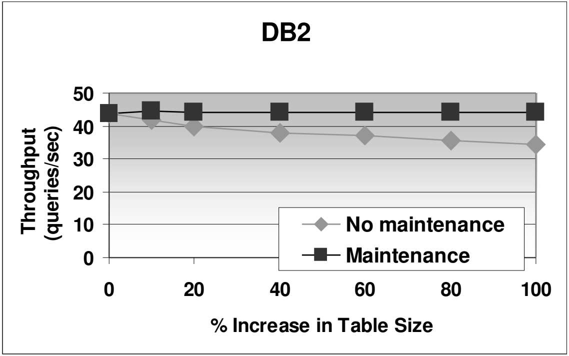
# Non-Covering vs. Table Scan




- query: range query
- non clustering: non-clustering non-covering index
- scan: no index, i.e., table scan required
- index is faster if less than 15% of the records are selected

DB2 UDB V7.1 Windows 2000

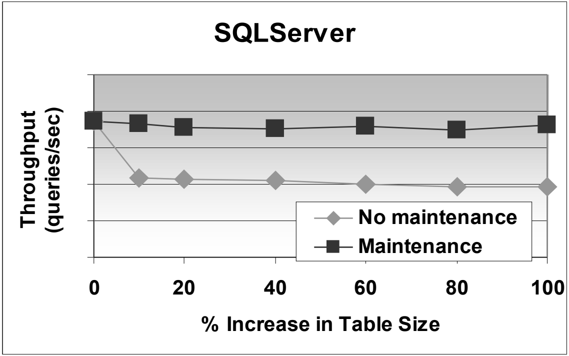
# Index Maintenance - DB2




- query: batch of 100 multi-point queries, `pctfree`=0 (data pages full)
- performance degrades with insertion
- overflow records simply appended
- query traverses index and then scans all overflow records
- reorganization helps

DB2 UDB V7.1 on Windows 2000

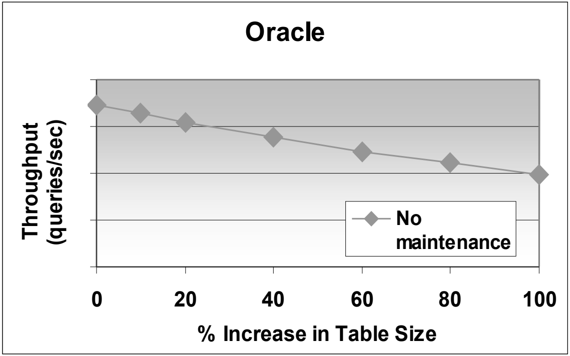
# Index Maintenance - SQL Server




- `fillfactor`=100 (data pages full)
- performance degrades with insertion
- overflow chain maintained for overflowing page
- extra disk access
- reorganization helps

SQL Server 7 on Windows 2000

# Index Maintenance - Oracle




- `pctfree` = 0 (data pages full), performance degrades with insertion
- all indexes in Oracle are non-clustering
- index-organized table is clustered by primary key
- recreating index does not reorganize table
- maintenance: export and re-import table to reorganize :-(

Oracle 8i EE on Windows 2000