ALMOST EVERYTHING AROUND DATABASE INDEXING

- Heap, B+Tree, Hashing
- Index Sequential Access Method
- Indexing Structure & Performance
- Clustered & Non-Clustered Indexing
- Deep Dive on B-Tree
- Covering Index

Based on slides from Shasha et al, Kifer et al.

INDEXING

Agenda

- Access Path
  - Type of queries
  - Heap vs. indexes
  - Clustered vs. Unclustered
  - Dense vs. Sparse
- Data Structures
  - ISAM
  - B+-Tree
  - Hash
- Tuning

Access Path

- Refers to the algorithm + data structure (*e.g.*, an index) used for retrieving and storing data in a table
- The choice of an access path to use in the execution of an SQL statement has no effect on the semantics of the statement
- This choice can have a major effect on the execution time of the statement

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

```
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;
```

Storage Structure

- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows in this case)
    - ISAM
    - B^+ tree
    - Hash

Heap Files

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Transcript Stored as a Heap File

666666	MGT123	F1994	4.0
123456	CS305	S1996	4.0
987654	CS305	F1995	2.0

page 0

717171	CS315	S1997	4.0
666666	EE101	S1998	3.0
765432	MAT123	S1996	2.0
515151	EE101	F1995	3.0

page 1

234567	CS305	S1999	4.0
878787	MGT123	S1996	3.0

page 2

Heap File - Performance

- Assume file contains *F* pages
- *Inserting a row*:
  - Access path is scan
  - Avg. *F*/2 page transfers if row already exists
  - *F*+1 page transfers if row does not already exist

Heap File - Performance

- Query
  - Access path is scan
  - Organization efficient if query returns all rows and order of access is not important

    SELECT * FROM Transcript
  - Organization inefficient if a *few* rows are requested
    - Average *F*/2 pages read to get get a single row

```
SELECT T.Grade
FROM Transcript T
WHERE T.StudId=12345 AND T.CrsCode = „CS305“
        AND T.Semester = „S2000“
```

Heap File - Performance

– Organization inefficient when a subset of rows is requested: F pages must be read

```
SELECT T.Course, T.Grade
FROM Transcript T                          -- point query
WHERE T.StudId = 123456

SELECT T.StudId, T.CrsCode
FROM Transcript T                          -- range query
WHERE T.Grade BETWEEN 2.0 AND 4.0
```

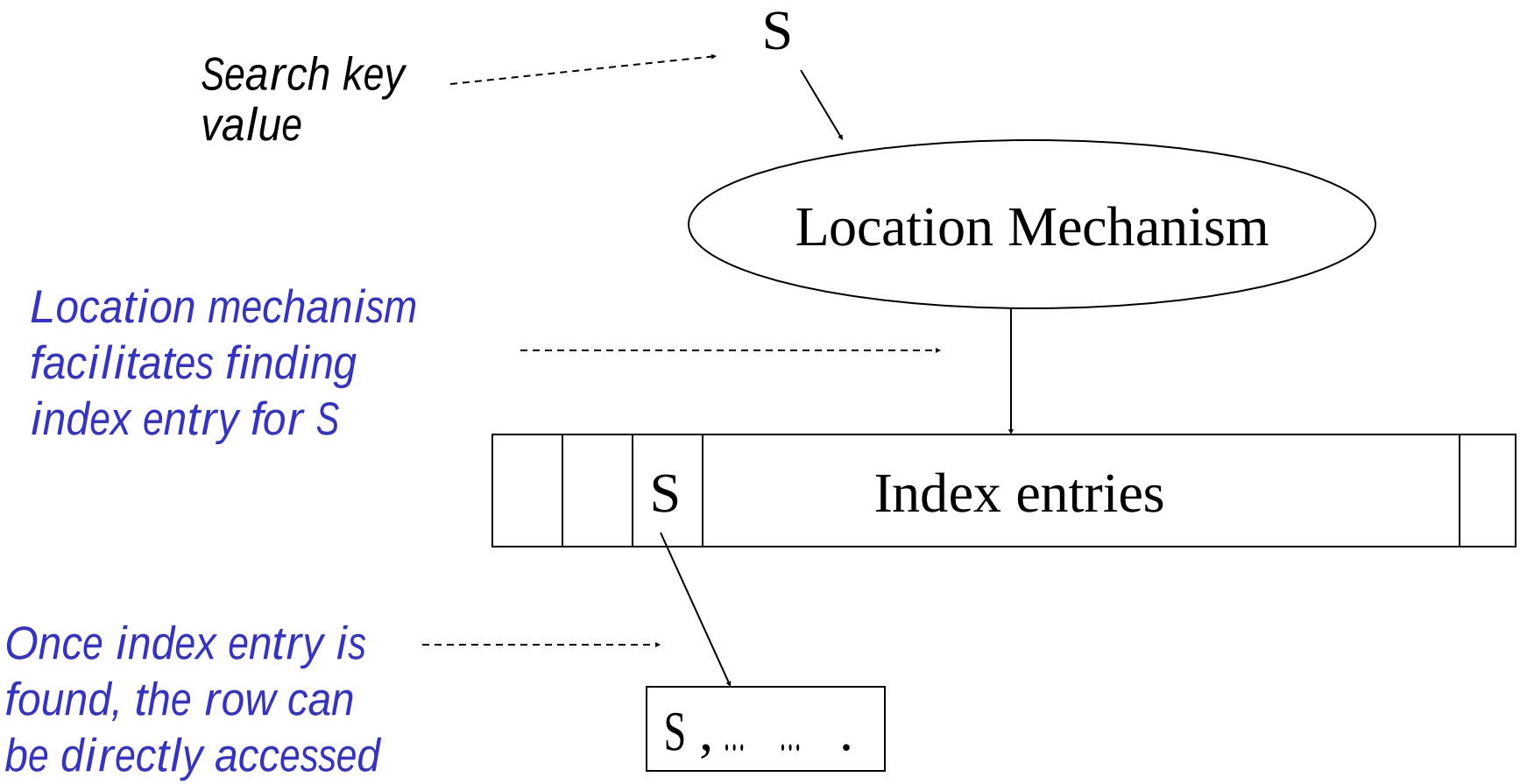
Index

- Mechanism for efficiently locating row(s) without having to scan entire table
- Based on a *search key:* rows having a particular value for the search key attributes can be quickly located
- Don't confuse candidate key with search key:
  - Candidate key: *set* of attributes; *guarantees* uniqueness
  - Search key: *sequence* of attributes; *does not guarantee* uniqueness – just used for search

Index Structure

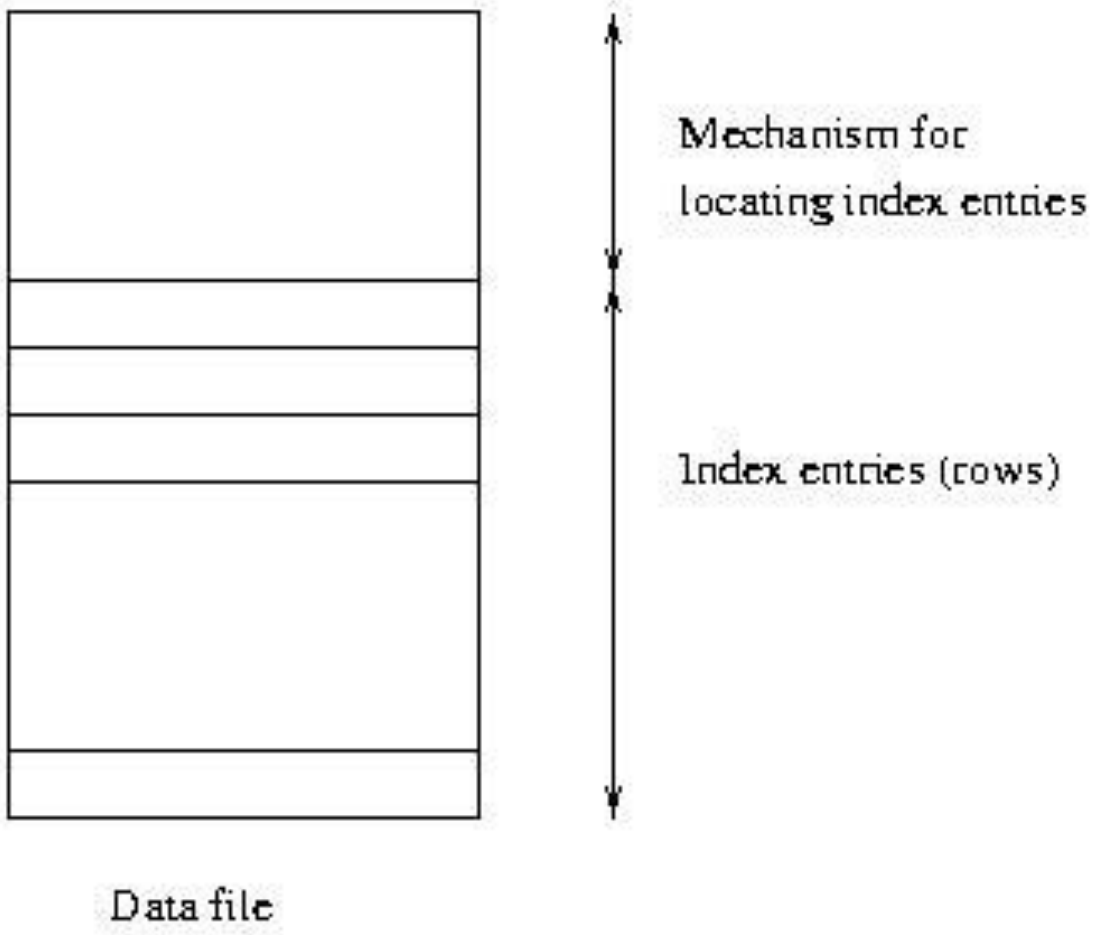
- Contains:
  - *Index entries*
    - Can contain the data tuple itself (index and table are *integrated* in this case); or
    - Search key value and a pointer to a row having that value; table stored separately in this case – *unintegrated* index
  - *Location mechanism*
    - Algorithm + data structure for locating an index entry with a given search key value
  - Index entries are stored in accordance with the search key value
    - Entries with the same search key value are stored together (hash, B-tree)
    - Entries may be sorted on search key value (B-tree)

Index Structure

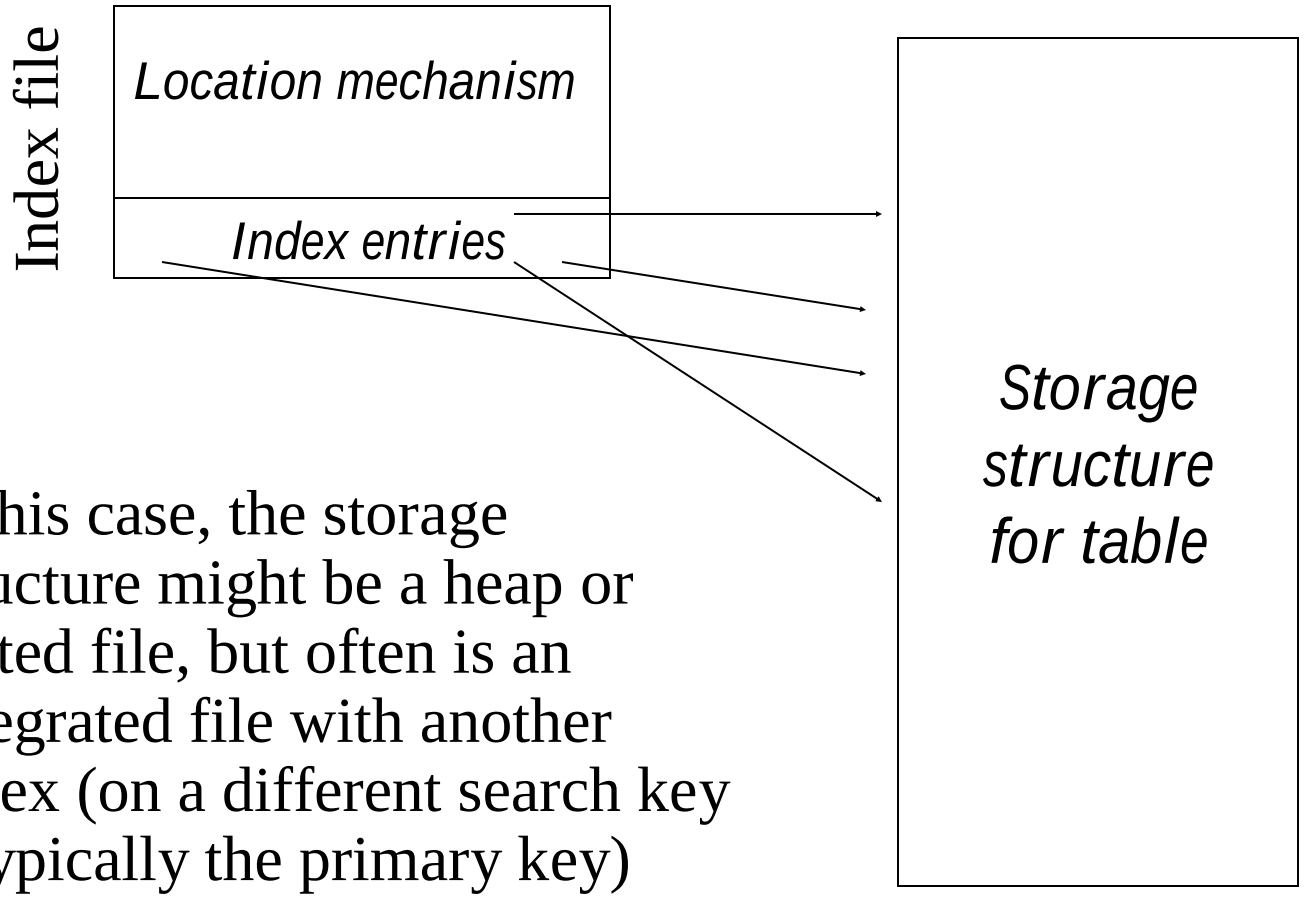


Integrated Storage Structure

Contains table and (main) index



Index File With Separate Storage Structure



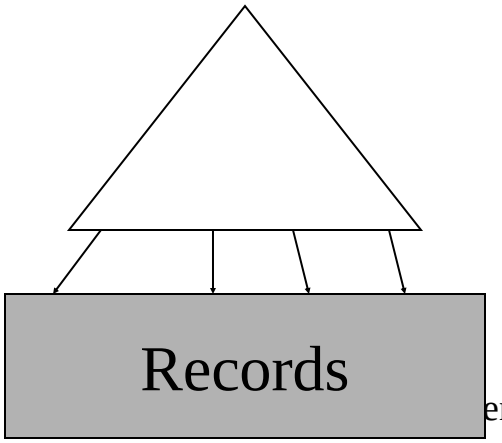
In this case, the storage structure might be a heap or sorted file, but often is an integrated file with another index (on a different search key – typically the primary key)

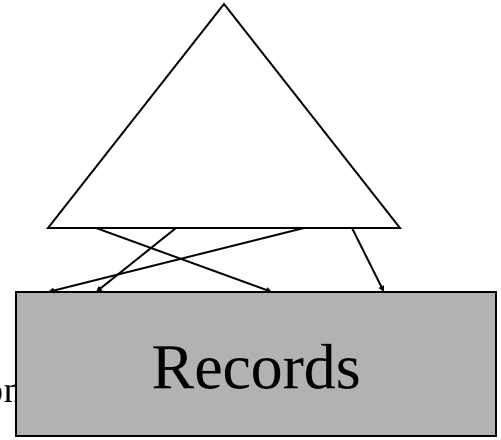
Indices: The Down Side

- Additional I/O to access index pages (except if index is small enough to fit in main memory)
- Index must be updated when table is modified.
- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - CREATE INDEX ind ON Transcript (*CrsCode*)
    - DROP INDEX ind

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.





Clustered Index

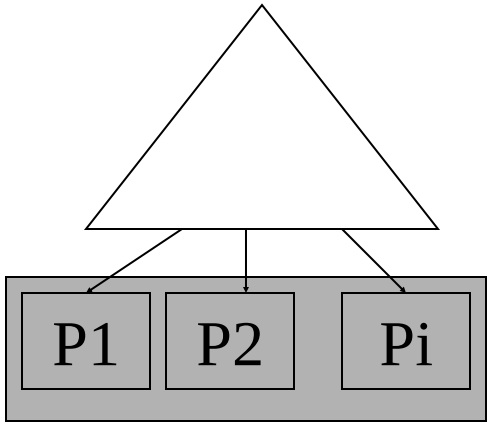
- Good for range searches when a range of search key values is requested
  - Use location mechanism to locate index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits

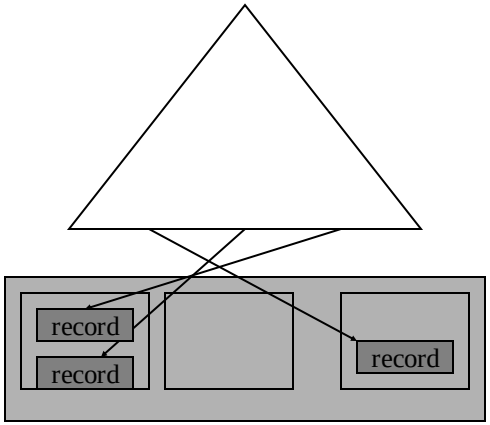
Example – Cost of Range Search

- Data file has 10,000 pages, 100 rows in search range
- Page transfers for table rows (assume 20 rows/page):
  - Heap: 10,000 (entire file must be scanned)
  - File sorted on search key: $\log_2$ 10000 + (5 or 6) 19
  - Unclustered index: 100
  - Clustered index: 5 or 6
- Page transfers for index entries (assume 200 entries/page)
  - Heap and sorted: 0
  - Unclustered secondary index: 1 or 2 (all index entries for the rows in the range must be read)
  - Clustered secondary index: 1 (only first entry must be read)

Dense / Sparse Index

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



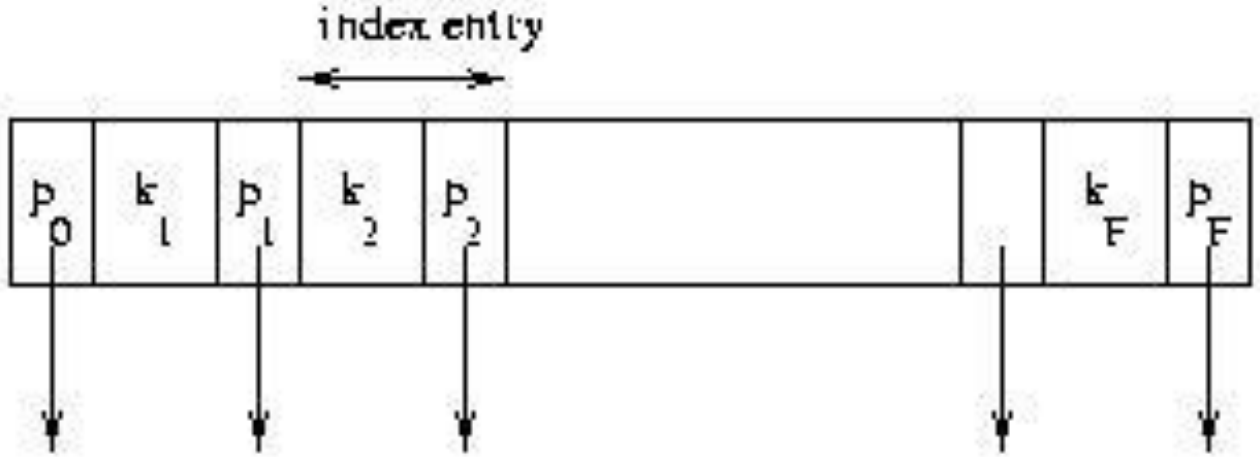


Agenda

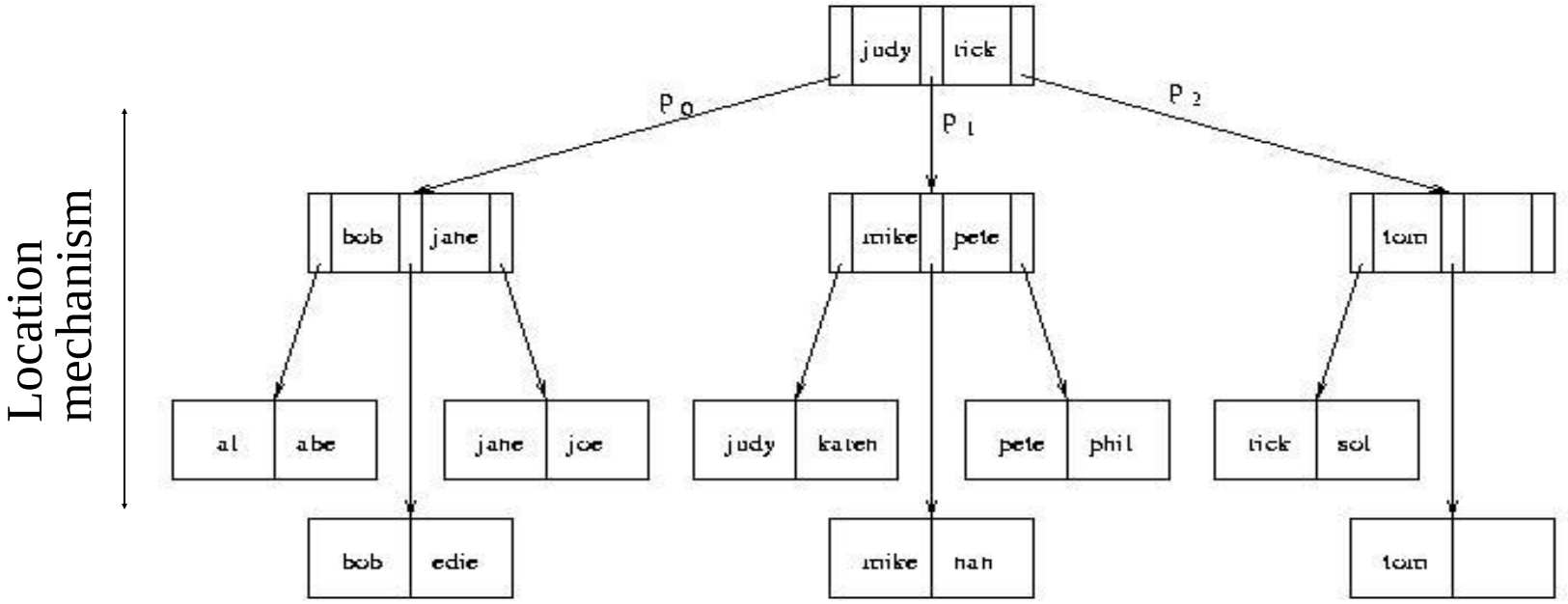
- Access Path
  - Type of queries
  - Heap vs. indexes
  - Clustered vs. Unclustered
  - Dense vs. Sparse
- Data Structures
  - ISAM
  - B+-Tree
  - Hash
- Tuning

Index Sequential Access Method (ISAM)

- Generally an integrated storage structure
  - Clustered, index entries contain rows
- Separator entry = (k_i , p_i); k_i is a search key value; p_i is a pointer to a lower level page
- k_i separates set of search key values in the two subtrees pointed at by p_{i-1} and p_i



Index Sequential Access Method

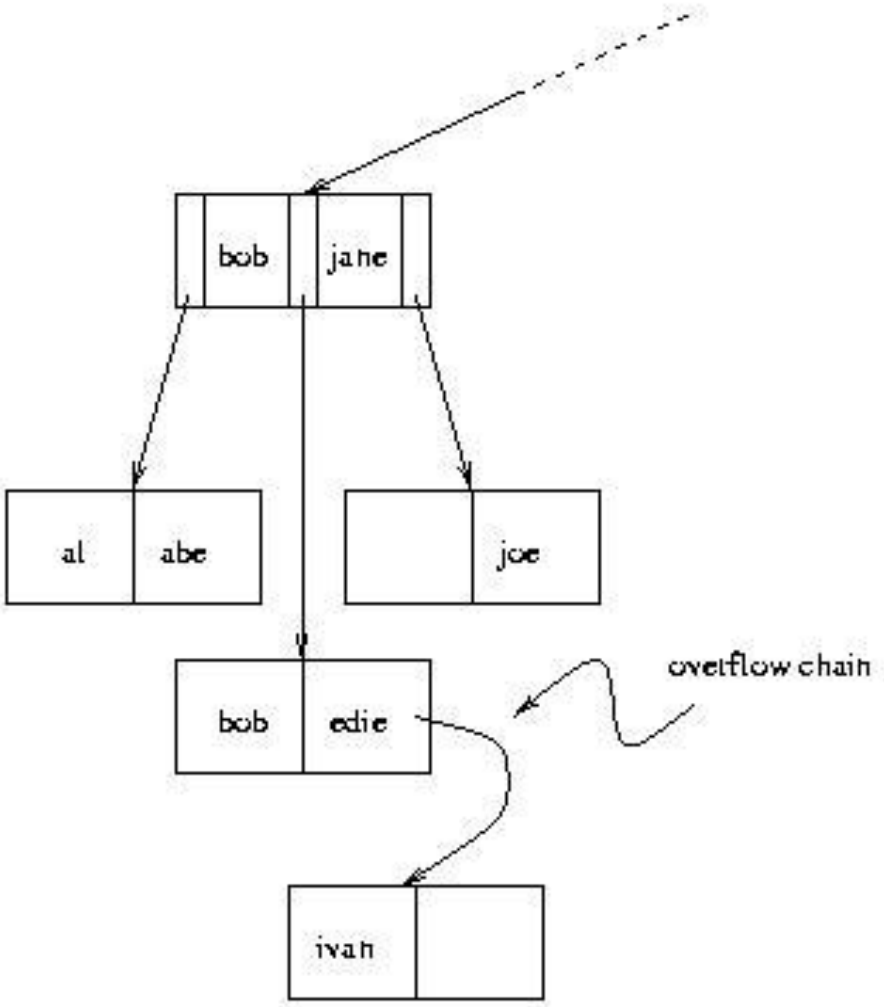


Index Sequential Access Method

- The index is static:
  - Once the separator levels have been constructed, they never change
  - Number and position of leaf pages in file stays fixed
- Good for equality and range searches
  - Leaf pages stored sequentially in file when storage structure is created to support range searches
    - if, in addition, pages are positioned on disk to support a scan, a range search can be very fast (static nature of index makes this possible)
- Supports multiple attribute search keys and partial key searches

Overflow Chains

- Contents of leaf pages change
  - Row deletion yields empty slot in leaf page
  - Row insertion can result in overflow leaf page and ultimately overflow chain
    - *Chains can be long, unsorted, scattered on disk*
    - *Thus ISAM can be inefficient if table is dynamic*

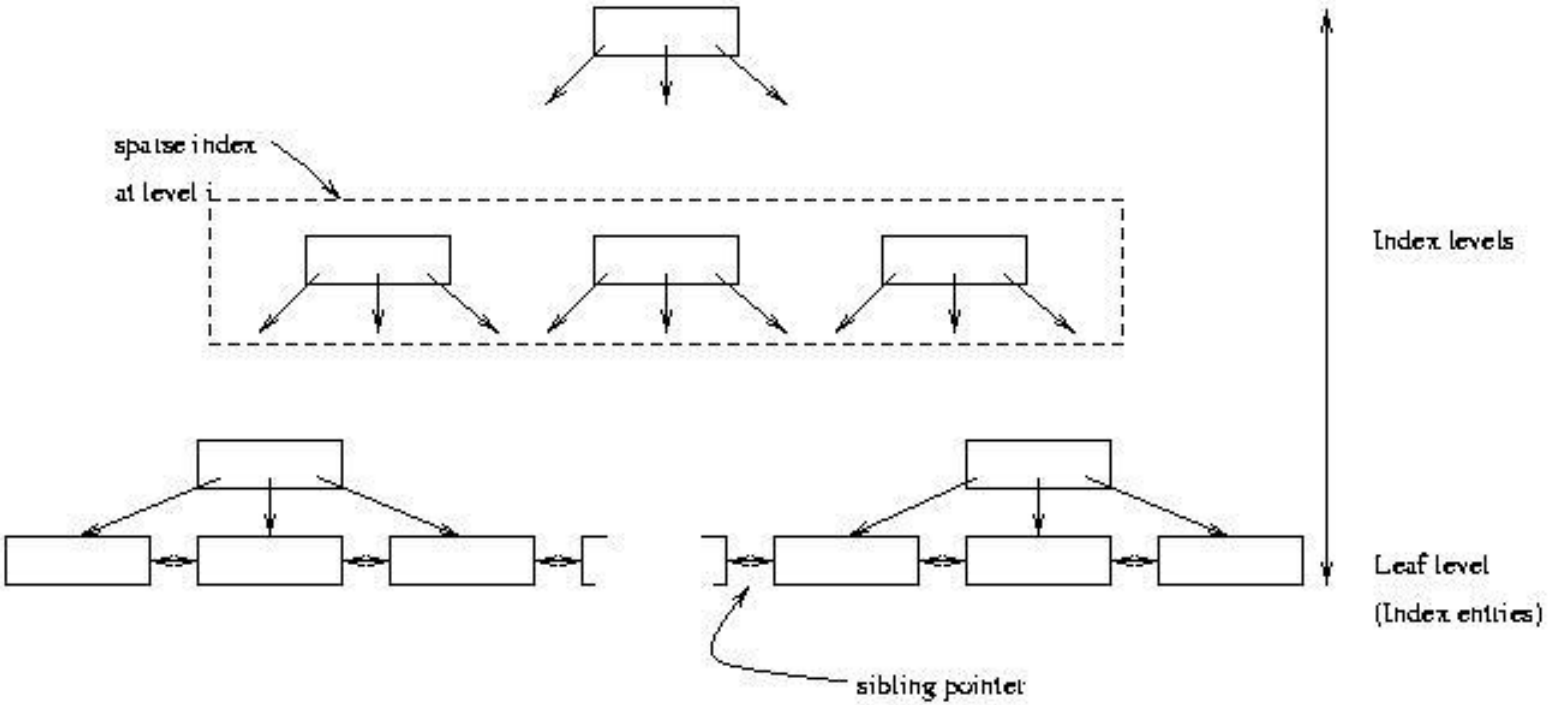


B^+ Tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure

➢ *Responds to dynamic changes in the table*

B^+ Tree Structure



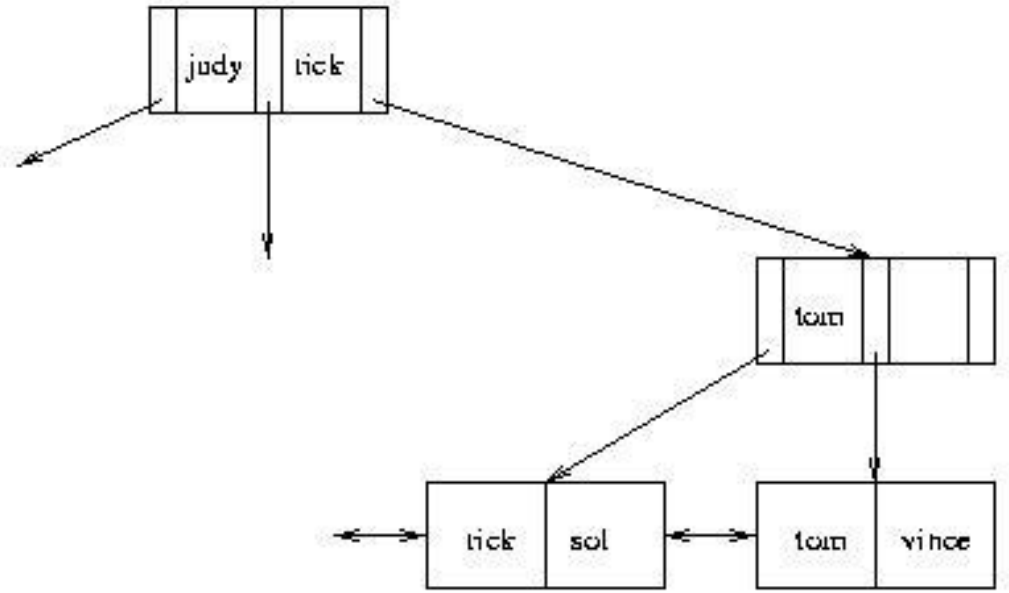
- Leaf level is a (sorted) linked list of index entries
- Sibling pointers support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous on disk)

Insertion and Deletion in B^+ Tree

- Structure of tree changes to handle row insertion and deletion – *no* overflow chains
- Tree remains *balanced*: all paths from root to index entries have same length
- Algorithm guarantees that the number of separator entries in an index page is between */2* and (is the fanout of a non leaf node)
  - Hence the maximum search cost is $log_{/2}Q + 1$ (with ISAM search cost depends on length of overflow chain)

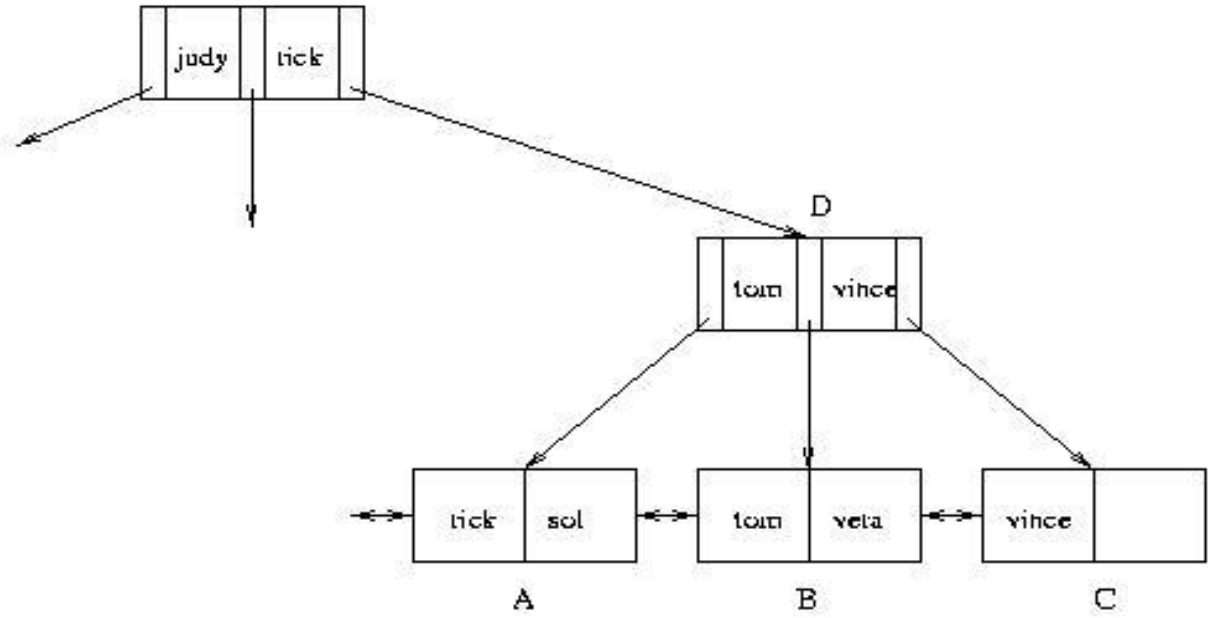
Handling Insertions - Example

- Insert "vince"



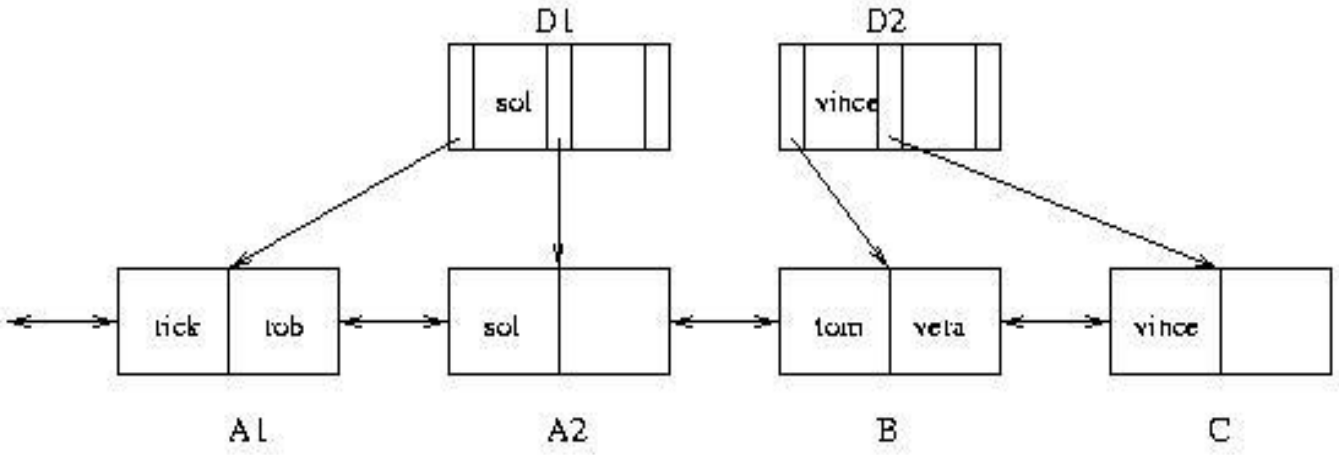
Handling Insertions (cont'd)

- Insert "vera": Since there is no room in leaf page:
  1. Create new leaf page, C
  2. Split index entries between B and C (but maintain sorted order)
  3. Add separator entry at parent level



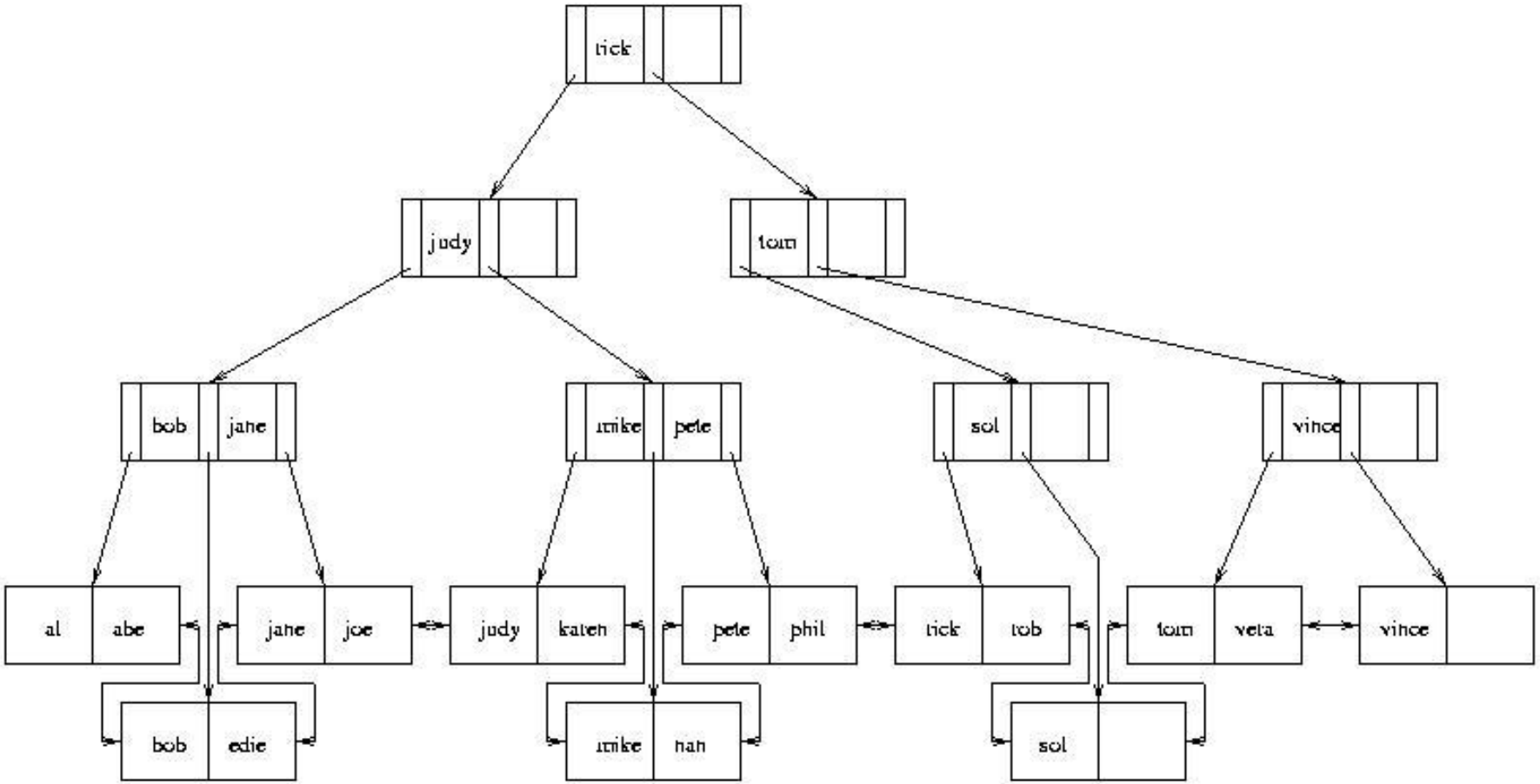
Handling Insertions (con't)

- Insert "rob". Since there is no room in leaf page A:
  1. Split A into A1 and A2 and divide index entries between the two (but maintain sorted order)
  2. Split D into D1 and D2 to make room for additional pointer
  3. Three separators are needed: "sol", "tom" and "vince"



Handling Insertions (cont'd)

- When splitting a separator page, push a separator up
- Repeat process at next level
- Height of tree increases by one



Handling Deletions

- Deletion can cause page to have fewer than */2* entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
  - See book
- In practice, tables generally grow, and merge algorithm is often not implemented
  - *Reconstruct tree to compact it*

Index Locks, Predicate Locks, and Key-Range Locking

- If a WHERE clause refers to a predicate name = mary and if there is an index on name, then an index lock on the index entries for name = mary is like a predicate lock on that predicate
- If a WHERE clause refers to a predicate such as 50000< salary < 70000 and if there is an index on salary, then a key-range index lock can be used to get the equivalent of a predicate lock on the predicate 50000<salary<70000

Key-Range Locking

- Instead of locking index pages, index entries at the leaf level are locked
  - Each such lock is interpreted as a lock on a range
- Suppose the domain of an attribute is A … Z and suppose at some time the entries in the index are

  C G P R X
- A lock on G is interpreted as a lock on the half-open interval

  [G P)
  - Which includes G but not P

Key-Range Locking (cont)

- Recall the index entries are: C G P R X
- Two special cases
  - A lock on X locks everything greater than X
  - A new lock must be provided for [A C)
- Then for example to lock the interval H < K < Q, we would lock G and P

Key-Range Locking (cont)

- Recall the index entries are: C G P R X
- To insert a new key, J, in the index
  - Lock G thus locking the interval [G P)
  - Insert J thus splitting the interval into [G J) [J P)
  - Lock J thus locking [J P)
  - Release the lock on G
- If a SELECT statement had a lock on G as part of a key-range, then the first step of the insert protocol could not be done
  - Thus phantoms are prevented and the key-range lock is equivalent to a predicate lock

Locking a B-Tree Index

- Many operations need to access an index structure concurrently
  - This would be a bottleneck if conventional two-phase locking mechanisms were used
- Because we understand the semantics of the index, we can develop a more efficient locking algorithm
  - The goal is to maintain isolation amount different operations that are concurrently accessing the index
  - The short term locks on the index structure are called latches
    - The long term locks on leaf entries we have been discussing are still obtained

Locking a B-Tree Index (cont)

- Read Locks
  - Obtain a read lock on the root, and work your way down the tree locking each entry as it is reached
  - When a new entry is locked, the lock on the previous entry (its parent) can be released
    - This operation will never revisit the parent
    - No write operation of a concurrent transaction can pass this operation as it goes down the tree
    - Called lock coupling or crabbing

Locking a B-Tree Index (cont)

- Write Locks
  - Obtain a write lock on the root, and work your way down the tree locking each entry as it is reached
  - When a new entry, n, is locked, if that entry is not full, the locks on all its parents can be released
    - An insert operation might have to go back up the tree, revisiting and perhaps splitting some nodes
    - Even if that occurs, because n is not full, it will not have to split n and therefore will not have to go further up the tree
    - Thus it can release locks further up in the tree.

Hash Index

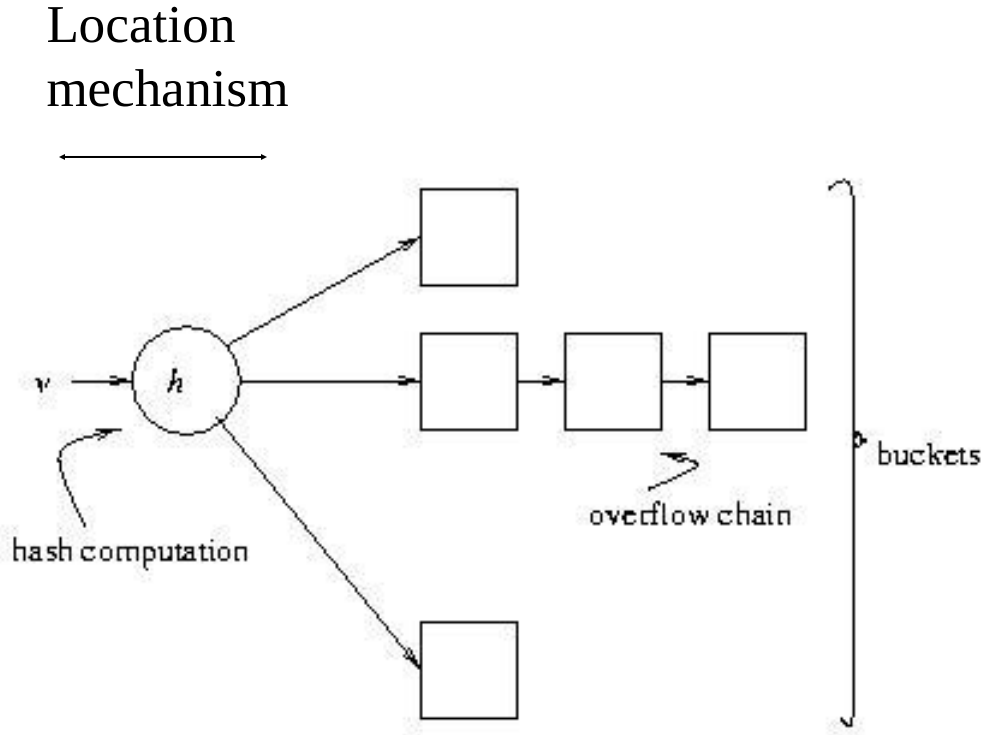
- Index entries partitioned into *buckets* in accordance with a *hash function*, *h(v)*, where *v* ranges over search key values
  - Each bucket is identified by an address, *a*
  - Bucket at address *a* contains all index entries with search key *v* such that *h(v) = a*
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

Equality Search with Hash Index

Given *v*:

1. Compute *h(v)*
2. Fetch bucket at *h(v)*
3. Search bucket

Cost = number of pages in bucket (cheaper than B^+ tree, if no overflow chains)



Hash Indices – Problems

- Does not support range search
  - Since adjacent elements in range might hash to different buckets, there is no efficient way to scan buckets to locate all search key values v between v_1 and v_2
- Although it supports multi-attribute keys, it does not support partial key search
  - Entire value of v must be provided to h
- Dynamically growing files produce overflow chains, which negate the efficiency of the algorithm

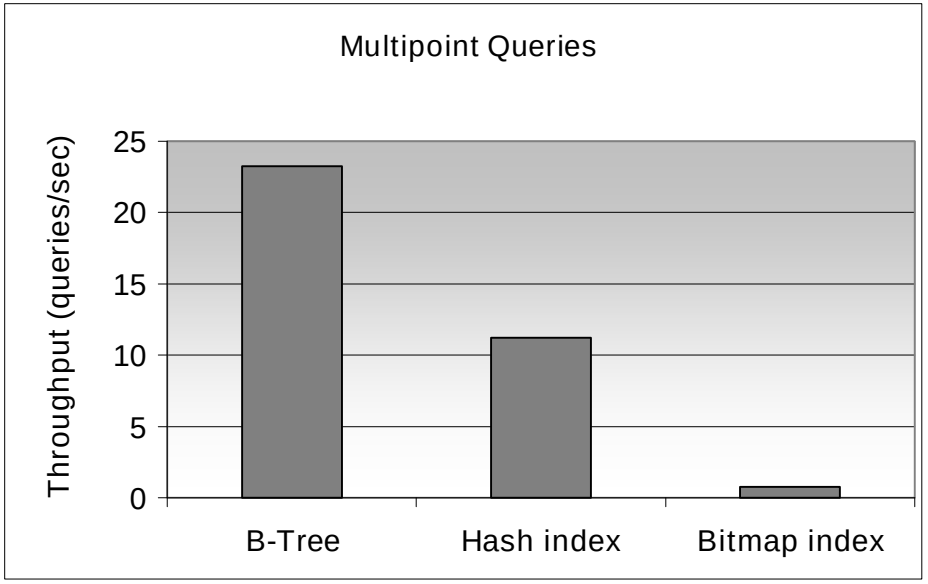
Agenda

- Access Path
  - Type of queries
  - Heap vs. indexes
  - Clustered vs. Unclustered
  - Dense vs. Sparse
- Data Structures
  - ISAM
  - B+-Tree
  - Hash
- Tuning

Index Tuning Knobs

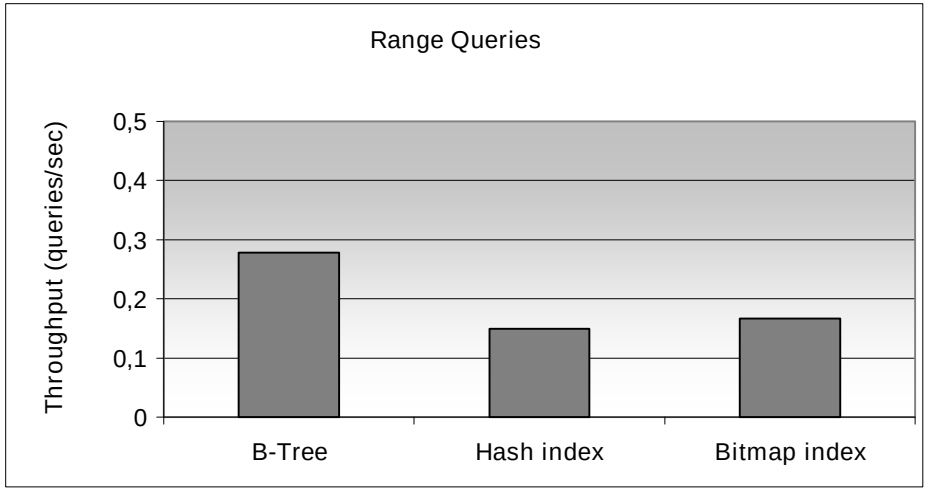
- Index data structure
- Search key
- Size of key
- Clustered/Non-clustered/No index
- Covering
- Maintenance

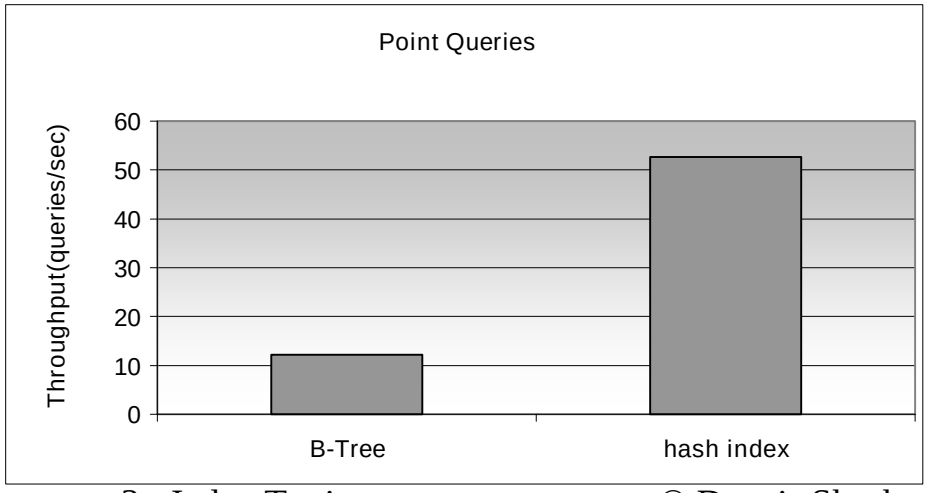
Multipoint query: B-Tree, Hash Tree



- There is an overflow chain in a hash index
- In a clustered B-Tree index records are on contiguous pages.

B-Tree, Hash Tree





- Hash indexes don't help when evaluating range queries
- Hash index outperforms B-tree on point queries

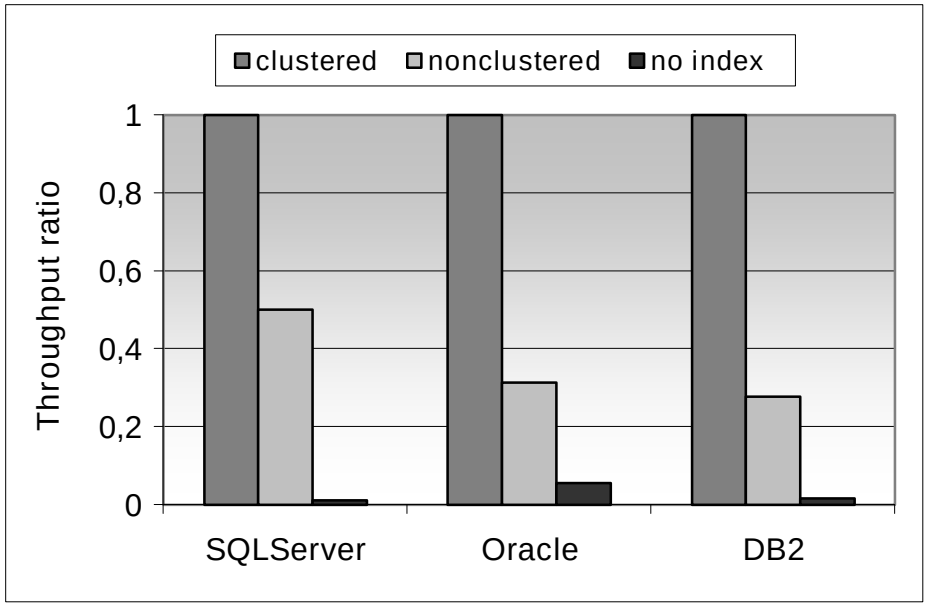
Key Compression

- 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 not CPU-bound
  - Updates are relatively rare

Clustered Index

- 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
  - Low insertion/update rate

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.

Non-Clustered Index

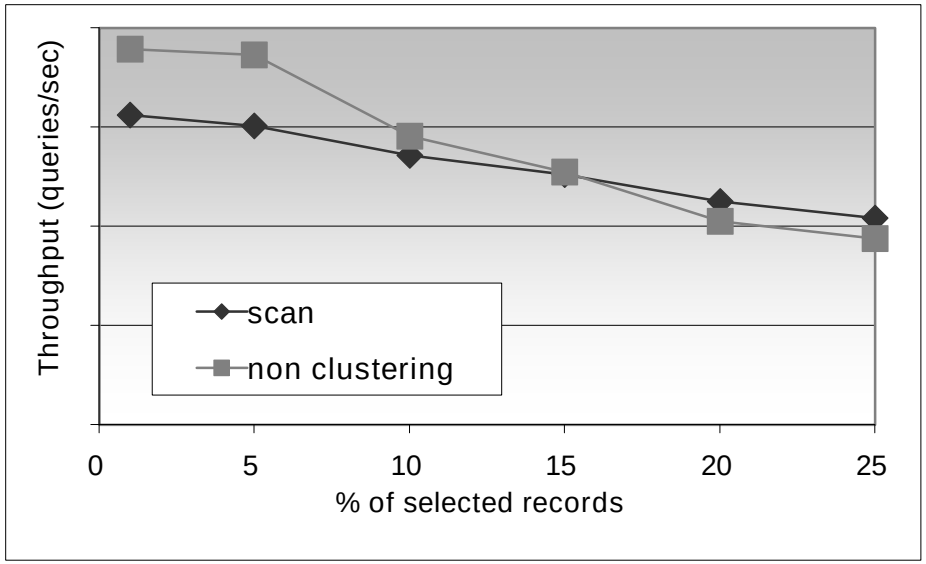
Benefits of non-clustered indexes

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
   - Multipoint queries:

*number of distinct key values > c * number of records per page*

Where c is the number of pages retrieved in each prefetch

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. Crossover > 10% when records are large or table is fragmented on disk – scan cost increases.

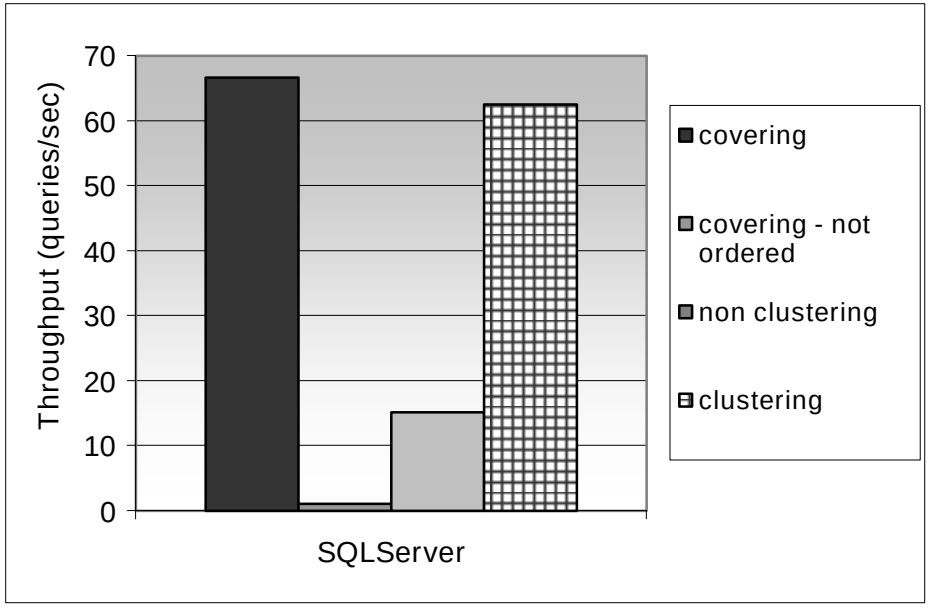
Multiple Attribute Search Key

- CREATE INDEX Inx ON Tbl (A*tt1*, A*tt2*)
- Search key is a *sequence* of attributes; index entries are lexically ordered
- Supports finer granularity equality search:
  - "Find row with value (A1, A2)"
- Supports range search (tree index only):
  - "Find rows with values between (A1, A2) and (A1, A2)"
- Supports partial key searches (tree index only):
  - Find rows with values of A*tt1* between A1 and A1
  - But not "Find rows with values of A*tt2* between A2 and A2"

Covering Index - defined

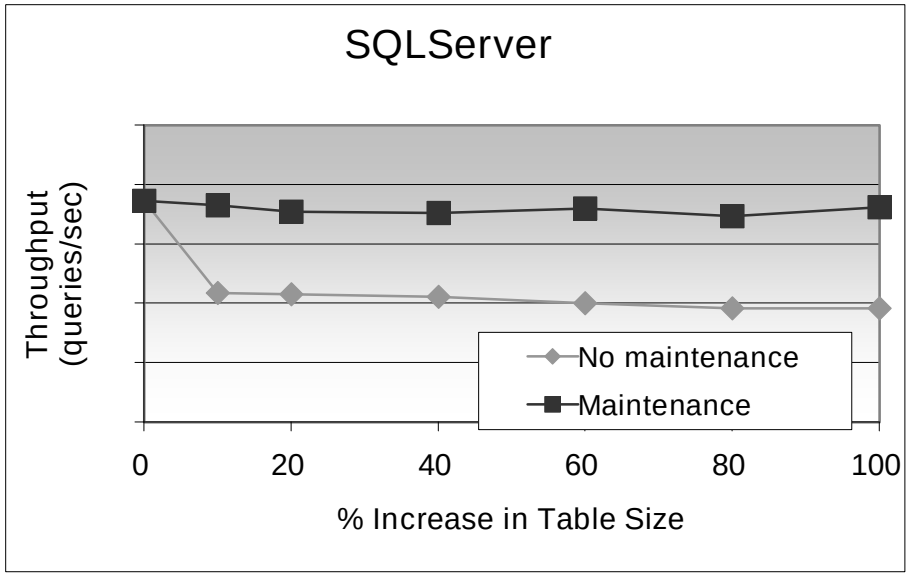
- Select name from employee where department = "marketing"
- Good covering index would be on (department, name)
- Index on (name, department) less useful.
- Index on department alone moderately useful.

Covering Index - impact



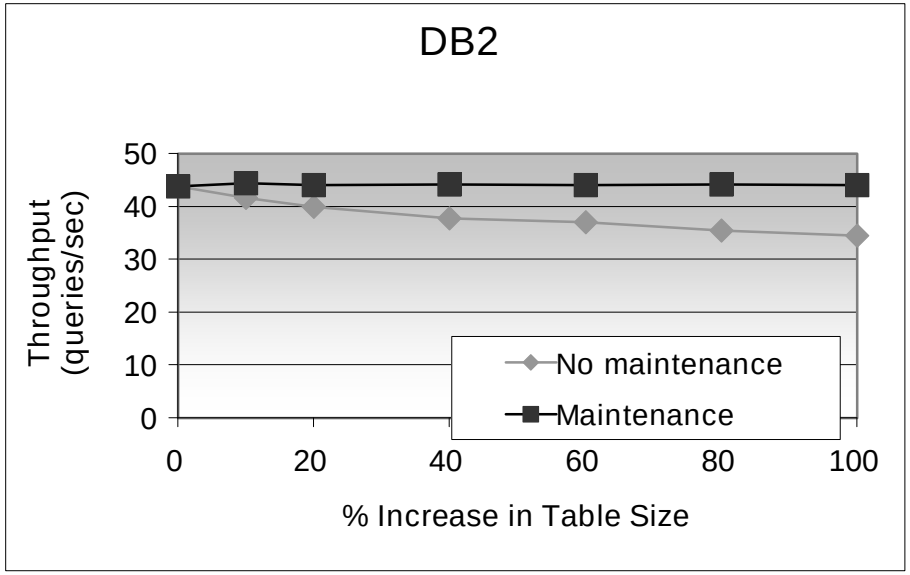
- Covering index performs better than clustering index when first attributes of index are in the where clause and last attributes in the select.
- When attributes are not in order then performance is much worse.

Index "Face Lifts"



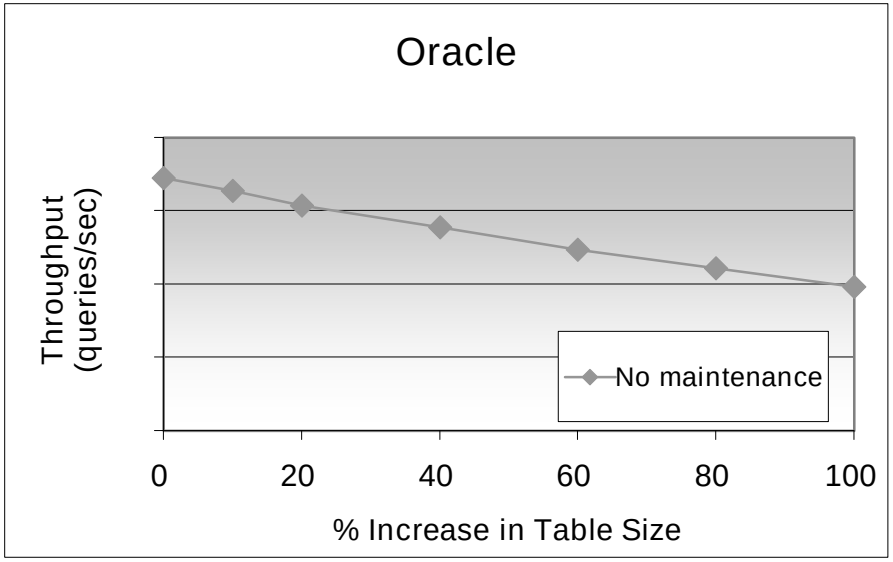
- Index is created with fillfactor = 100.
- Insertions cause page splits and extra I/O for each query
- Maintenance consists in dropping and recreating the index
- With maintenance performance is constant while performance degrades significantly if no maintenance is performed.

Index "Face Lifts"



- 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.

Index “Face lifts”

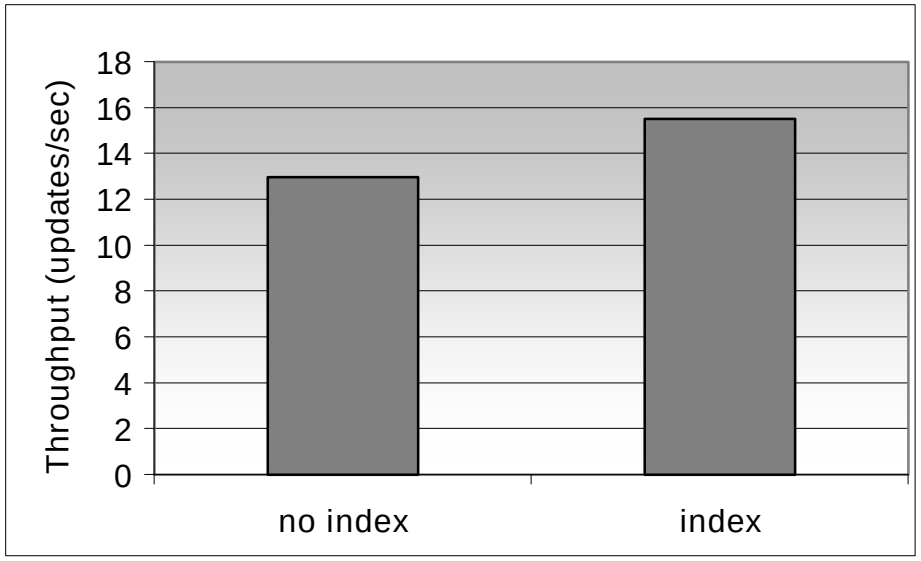


- In Oracle, clustered index are approximated by an index defined on a clustered table
- No automatic physical reorganization
- Index defined with pctfree = 0
- Overflow pages cause performance degradation

Index on Small Tables

- Tuning manuals suggest to avoid indexes on small tables
  - If all data from a relation fits in one page then an index page adds an I/O
  - If each record fits in a page then an index helps performance

Index on Small Tables



- 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 allow to take advantage of row locking.

Summary

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 records returned
   - 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