Database Tuning

Index Tuning

Nikolaus Augsten

University of Salzburg
Department of Computer Science
Database Group

Unit 6 – WS 2014/2015

Adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

1. Index Tuning
   - Indexes and Joins
   - Index Tuning Examples

Outline

1. Index Tuning
   - Indexes and Joins
   - Index Tuning Examples

Join Strategies – Running Example

- Relations: R and S
  - disk block size: $4kB$
  - R: $n_r = 5000$ records, $b_r = 100$ disk blocks, $0.4MB$
  - S: $n_s = 10000$ records, $b_s = 400$ disk blocks, $1.6MB$
- Running Example: $R \bowtie S$
  - R is called the outer relation
  - S is called the inner relation

Example from *Silberschatz, Korth, Sudarashan. Database System Concepts. McGraw-Hill.*

Join Strategies – Naive Nested Loop

- Naive nested loop join
  - take each record of R (outer relation) and search through all records of S (inner relation) for matches
  - for each record of R, S is scanned
- Example: Naive nested loop join
  - worst case: buffer can hold only one block of each relation
  - R is scanned once, S is scanned n_r times
  - in total $n_r b_s + b_r = 2,000,100$ blocks must be read $(= 8GB)$!
  - note: worst case different if S is outer relation
  - best case: both relations fit into main memory
  - $b_s + b_r = 500$ block reads

Join Strategies – Block Nested Loop

- Block nested loop join
  - compare all rows of each block of R to all records in S
  - for each block of R, S is scanned
- Example: (continued)
  - worst case: buffer can hold only one block of each relation
  - R is scanned once, S is scanned b_r times
  - in total $b_r b_s + b_r = 40,100$ blocks must be read ($= 160MB$)
  - best case: $b_s + b_r = 500$ block reads

Join Strategies – Indexed Nested Loop

- Indexed nested loop join
  - take each row of R and look up matches in S using index
  - runtime is $O(|R| \times \log |S|)$ (vs. $O(|R| \times |S|)$ of naive nested loop)
  - efficient if index covers join (no data access in S)
  - efficient if R has less records than S has pages: not all pages of S must be read (e.g., foreign key join from small to large table)
- Example: (continued)
  - B^+-tree index on S has 4 layers, thus max. $c = 5$ disk accesses per record of S
  - in total $b_r + n_r c = 25,100$ blocks must be read ($= 100MB$)

Join Strategies – Merge Join

- Merge join (two clustered indexes)
  - scan R and S in sorted order and merge
  - each block of R and S is read once
- No index on R and/or S
  - if no index: sort and store relation with $b(2\lceil log_{M-1}(b/M)\rceil + 1) + b$ block transfers (M: free memory blocks)
  - if non-clustered index present: index scan possible
- Example: (continued)
  - best case: clustered indexes on R and S ($M = 2$ enough)
  - $b_r + b_s = 500$ blocks must be read ($2MB$)
  - worst case: no indexes, only $M = 3$ memory blocks
  - sort and store R (1400 blocks) and S (7200 blocks) first: join with 9100 ($36MB$) block transfers in total
  - case $M = 25$ memory blocks: 2500 block transfers ($10MB$)

Join Strategies – Hash Join

- Hash join (equality, no index):
  - hash both tables into buckets using the same hash function
  - join pairs of corresponding buckets in main memory
  - R is called probe input, S is called build input
- Joining buckets in main memory:
  - build hash index on one bucket from S (with new hash function)
  - probe hash index with all tuples in corresponding bucket of R
  - build bucket must fit main memory, probe bucket needs not
- Example: (continued)
  - assume that each probe bucket fits in main memory
  - R and S are scanned to compute buckets, buckets are written to disk, then buckets are read pairwise
  - in total $3(b_r + b_s) = 1500$ blocks are read/written ($6MB$)
  - default in SQLServer and DB2 UDB when no index present

Distinct Values and Join Selectivity

- Join selectivity:
  - number of retrieved pairs divided by cardinality of cross product ($|R \bowtie S|/|R \times S|$)
  - selectivity is low if join result is small
- Distinct values refer to join attributes of one table
- Performance decreases with number of distinct join values
  - few distinct values in both tables usually means many matching records
  - many matching records: join result is large, join slow
  - hash join: large buckets (build bucket does not fit main memory)
  - index join: matching records on multiple disk pages
  - merge join: matching records do not fit in memory at the same time

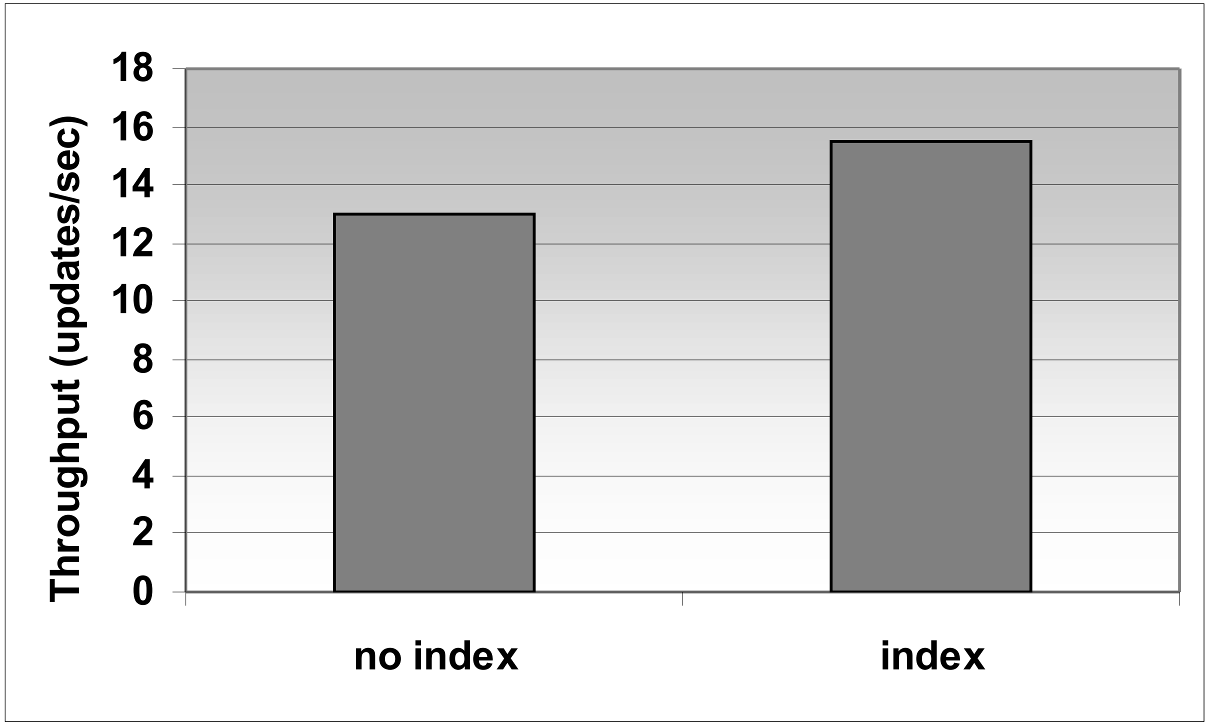
Foreign Keys

- Foreign key: attribute $R.A$ stores key of other table, $S.B$
- Foreign key constraints: $R.A$ must be subset of $S.B$
  - insert in R checks whether foreign key exists in S
  - deletion in S checks whether there is a record with that key in R
- Index makes checking foreign key constraints efficient:
  - index on $R.A$ speeds up deletion from S
  - index on $S.B$ speeds up insertion into R
  - some systems may create index on $R.A$ and/or $S.B$ by default
- Foreign key join:
  - each record of one table matches at most one record of the other table
  - most frequent join in practice
  - both hash and index nested loop join work well

Indexes on Small Tables

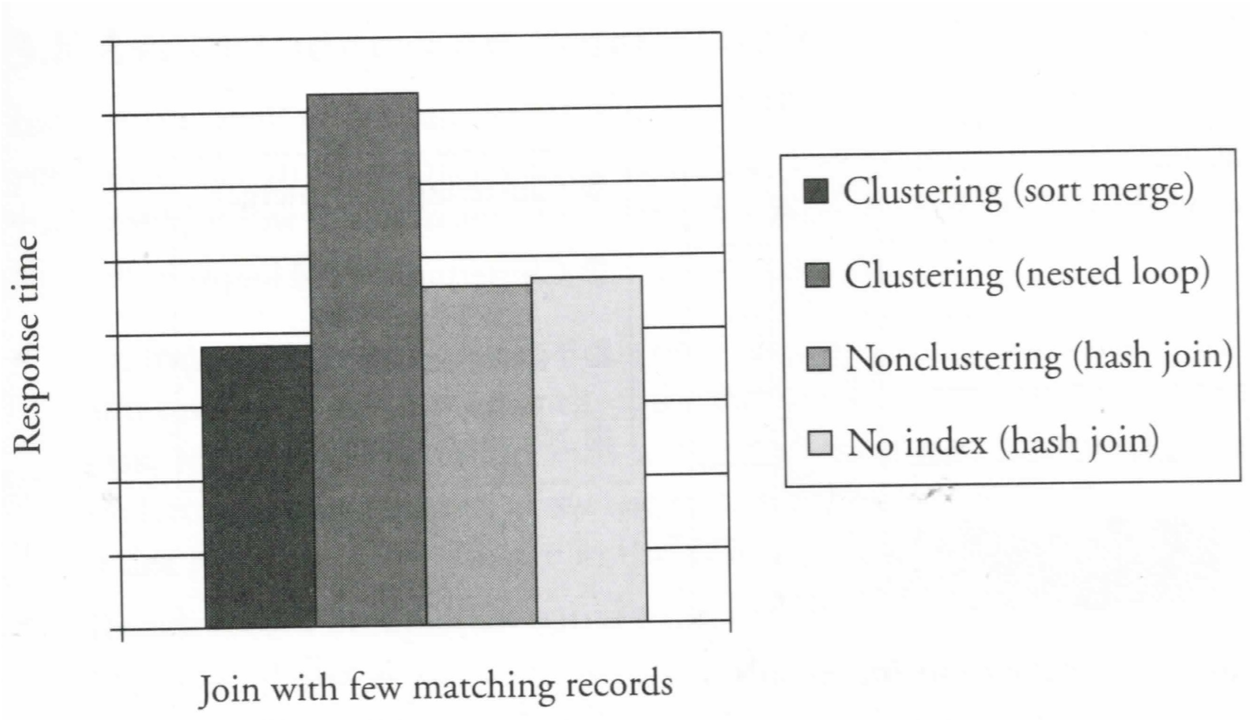
- Read query on small records:
  - tables may fit on a single track on disk
  - read query requires only one seek
  - index not useful: seeks at least one index page and one table page
- Table with large records (∼page size):
  - each record occupies a whole page
  - for example, 200 records occupy 200 pages
  - index useful for point queries (read 3 pages vs. 200)
- Many inserts and deletions:
  - index must be reorganized (locking!)
  - lock conflicts near root since index is small
- Update of single records:
  - without index table must be scanned
  - scanned records are locked
  - scan (an thus lock contention) can be avoided with index

Update Queries on a Small Tables



- Index avoids tables scan and thus lock contention.

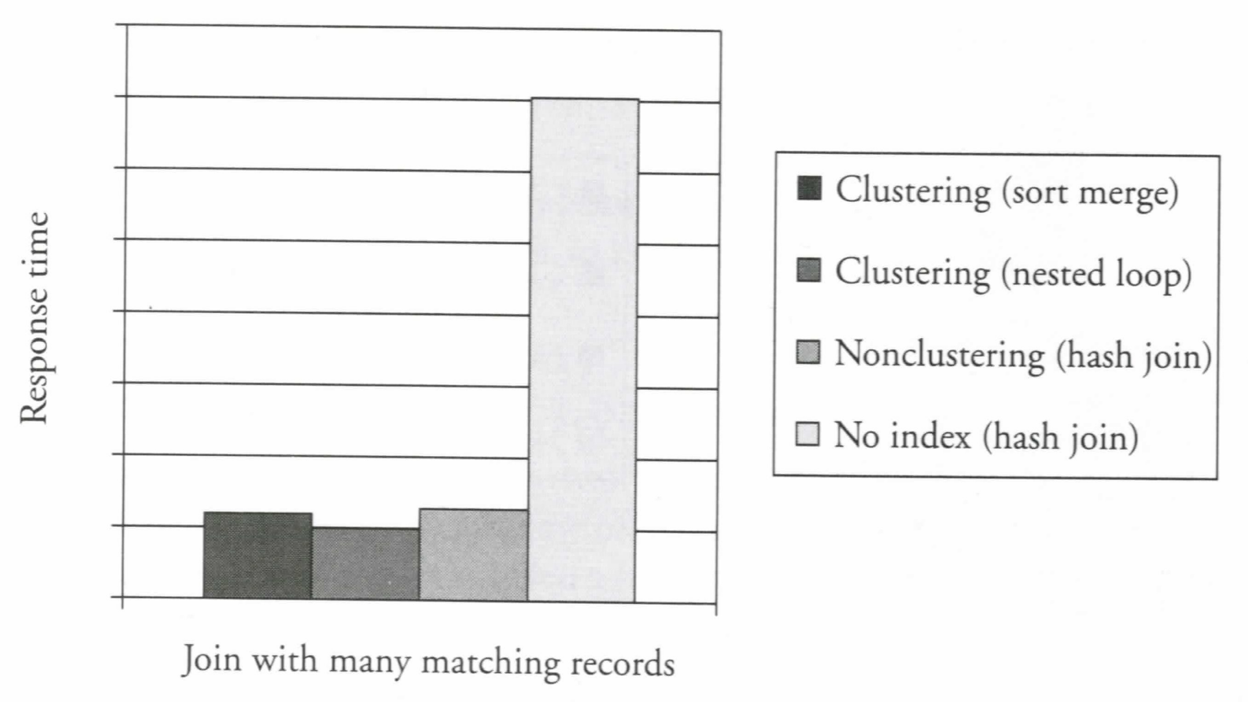
Experiment – Join with Few Matching Records



- non-clustered index is ignored, hash join used instead

SQL Server 7 on Windows 2000

Experiment – Join with Many Matching Records



- all joins slow since output size is large
- hash join (no index) slow because buckets are very large

SQL Server 7 on Windows 2000

Outline

1. Index Tuning
   - Indexes and Joins
   - Index Tuning Examples

Index Tuning Examples

- The examples use the following tables:
  - `Employee(ssnum,name,dept,manager,salary)`
  - `Student(ssnum,name,course,grade,stipend,evaluation)`

Exercise 1 – Query for Student by Name

- `Student` was created with non-clustering index on `name`.
- Query:

```
SELECT *
FROM Student
WHERE name='Bayer'
```

- Problem: Query does not use index on `name`.
- Solution: Try updating the catalog statistics.
  - Oracle, Postgres: `ANALYZE`
  - SQL Server: sp_createstats
  - DB2: `RUNSTATS`

Exercise 2 – Query for Salary I

- Non-clustering index on `salary`.
- Catalog statistics are up-to-date.
- Query:

```
SELECT *
FROM Emplyee
WHERE salary/12 = 4000
```

- Problem: Query is too slow.
- Solution: Index not used because of the arithmetic expression.
  Two Options:
  - Rewrite query:

    ```
    SELECT *
    FROM Emplyee
    WHERE salary = 48000
    ```

  - Use function based index.

Exercise 3 – Query for Salary II

- Non-clustering index on `salary`.
- Catalog statistics are up-to-date.
- Query:

```
SELECT *
FROM Emplyee
WHERE salary = 48000
```

- Problem: Query still does not use index. What could be the reason?
- Solution: The index is non-clustering. Many employees have a salary of 48000, thus the index may not help. It may still help for other, less frequent, salaries!

Exercise 4 – Clustering Index and Overflows

- Clustering index on `Student.ssnum`
- Page size: $2kB$
- Record size in `Student` table: $1KB$ (`evaluation` is a long text)
- Problem: Overflow when new evaluations are added.
- Solution: Clustering index does not help much due to large record size. A non-clustering index avoids overflows.

Exercise 5 – Non-clustering Index I

- `Employee` table:
  - 30 employee records per page
  - each employee belongs to one of 50 departments (`dept`)
  - the departments are of similar size
- Query:

```
SELECT ssnum
FROM Emplyee
WHERE dept = 'IT'
```

- Problem: Does a non-clustering index on `Employee.dept` help?
- Solution: Only if the index covers the query.
  - 30/50=60% of the pages will have a record with `dept = 'IT'`
  - table scan is faster than accessing 3/5 of the pages in random order

Exercise 6 – Non-clustering Index II

- `Employee` table:
  - 30 employee records per page
  - each employee belongs to one of 5000 departments (`dept`)
  - the departments are of similar size
- Query:

```
SELECT ssnum
FROM Emplyee
WHERE dept = 'IT'
```

- Problem: Does a non-clustering index on `Employee.dept` help?
- Solution: Only if the index covers the query.
  - only 30/5000=0.6% of the pages will have a record with `dept='IT'`
  - table scan is slower

Exercise 7 – Statistical Analysis

- Auditors run a statistical analysis on a copy of `Emplyee`.
- Queries:
  - count employees with a certain salary (frequent)
  - find employees with maximum or minimum salary within a particular department (frequent)
  - find an employee by its social security number (rare)
- Problem: Which indexes to create?
- Solution:
  - non-clustering index on `salary` (covers the query)
  - clustering composite index on `(dept, salary)` using a B^+-tree (all employees with the maximum salary are on consecutive pages)
  - non-clustering hash index on `ssnum`

Exercise 8 – Algebraic Expressions

- Student stipends are monthly, employee salaries are yearly.
- Query: Which employee is paid as much as which student?
- There are two options to write the query:

```
SELECT *
FROM Employee, Student
WHERE salary = 12*stipend
```

```
SELECT *
FROM Employee, Student
WHERE salary/12 = stipend
```

- Index on a table with an algebraic expression not used.
- Problem: Which query is better?

Exercise 8 – Solution

- If index on only one table, it should be used.
- Index on both tables, clustering on larger table: use it.
- non-clustering index on both tables:
  - use index on larger table
  - if the number of tuples in the small table is larger than the number of blocks in the large table, the system might decide not to use the index

Exercise 9 – Purchasing Department

- Purchasing department maintains table `Onorder(supplier,part,quantity,price)`.
- The table is heavily used during the opening hours, but not over night.
- Queries:
  - Q1: add a record, all fields specified (very frequent)
  - Q2: delete a record, `supplier` and `part` specified (very frequent)
  - Q3: find total quantity of a given part on order (frequent)
  - Q4: find the total value on order to a given supplier (rare)
- Problem: Which indexes should be used?

Exercise 9 – Solution

- Queries:

  Q1: add a record, all fields specified (very frequent)
  Q2: delete a record, `supplier` and `part` specified (very frequent)
  Q3: find total quantity of a given part on order (frequent)
  Q4: find the total value on order to a given supplier (rare)

- Solution: Clustering composite B^+-tree index on `(part,supplier)`.
  - eliminate overflows over night
  - attribute order important to support query Q3
  - hash index will not work for query Q3 (prefix match query)
- Discussion: Non-clustering index on `supplier` to answer query Q4?
  - index must be maintained and will hurt the performance of much more frequent queries Q1 and Q2
  - index does not help much if there are only few different suppliers

Exercise 10 – Point Query Too Slow

- `Employee` has a clustering B^+-tree index on `ssnum`.
- Queries:
  - retrieve employee by social security number (`ssnum`)
  - update employee with a specific social security number
- Problem: Throughput is still not enough.
- Solution: Use hash index instead of B^+-tree (faster for point queries).

Exercise 11 – Historical Immigrants Database

- Digitalized database of US immigrants between 1800 and 1900:
  - 17M records
  - each record has approx. 200 fields
    e.g., last name, first name, city of origin, ship taken, etc.
- Queries retrieve immigrants:
  - by last name and at least one other attribute
  - second attribute is often first name (most frequent) or year
- Problem: Efficiently serve 2M descendants of the immigrants. . .

Exercise 11 – Solution

- Clustering B^+-tree index on (`lastname,firstname`):
  - no overflow since database does not have updates
  - use high fill factor to increase space utilization
  - key compression should be used (long key, no update)
  - index useful also for prefix queries on `lastname`
- Composite non-clustering index on (`lastname,year`):
  - no maintenance cost (no updates)
  - attributes probably selective enough
- Non-clustering indexes on all frequent attribute combinations?
  - no maintenance, thus only limitation is space overhead
  - useful only if selective enough

Exercise 12 – Flight Reservation System

- An airline manages 1000 flights and uses the tables:
  - `Flight(flightID, seatID, passanger-name)`
  - `Totals(flightID, number-of-passangers)`
- Query: Each reservation
  - adds a record to `Flight`
  - increments `Totals.number-of-passangers`
- Queries are separate transactions.
- Problem: Lock contention on `Totals`.
- Solution:
  - Totals is a small table (1000 small records) and fits on few pages.
  - Without index, update scans table and scanned records are locked.
  - Clustering index on `flightID` avoids table scan and thus lock contention (row locking assumed).