Database Tuning Index Tuning

Nikolaus Augsten

University of Salzburg Department of Computer Science Database Group

Unit 3 – WS 2015/16

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

Outline



- Query Types
- Index Types
- Data Structures
- Composite Indexes
- Indexes and Joins
- Index Tuning Examples

Outline



• Query Types

- Index Types
- Data Structures
- Composite Indexes
- Indexes and Joins
- Index Tuning Examples



• 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
```

- 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'

• 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

- 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

Outline



- Query Types
- Index Types
- Data Structures
- Composite Indexes
- Indexes and Joins
- Index Tuning Examples



- 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 relational theory
 - relational 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 index node *n*
 - a new node *n'* must be allocated on disk
 - B^+ -tree: split *n* into *n* and *n'*, both at same distance from root
 - hash index: n stores pointer to new node n' (overflow chaining)

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:

ptrs in dense index = records per page \times 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-only query within index structure
 - fast: data records are 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:
 covered query, but not prefix:
 SELECT B, C
 FROM R
 WHERE A = 5
 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!)
- 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 → B → C, splitting B results in A → B' → B'' → 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 good if query not too selective
 - #*r*: number of records returned by query
 - #p: number of disk pages in table
 - the #r records are uniformly distributed over all pages
 - thus query will read min(#r, #p) 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 #r should be significantly smaller than #p

Non-Clustering Index and Multi-point Queries – Example

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

Index Tuning Index Types

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



- 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



- 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



- 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



- overflow chain maintained for overflowing page
- extra disk access
- reorganization helps

SQL Server 7 on Windows 2000



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

```
Oracle 8i EE on Windows 2000
```

Outline



- Query Types
- Index Types

• Data Structures

- Composite Indexes
- Indexes and Joins
- Index Tuning Examples

Index Data Structures

- Indexes can be implemented with different data structures.
- We discuss:
 - *B*⁺-tree index
 - hash index
 - bitmap index (briefly)
- Not discussed here:
 - dynamic hash indexes: number of buckets modified dynamically
 - R-tree: index for spatial data (points, lines, shapes)
 - quadtree: recursively partition a 2D plane into four quadrants
 - octree: quadtree version for three dimensional data
 - main memory indexes: T-tree, 2-3 tree, binary search tree



- balanced tree of key-pointer pairs
- keys are sorted by value
- nodes are at least half full
- access records for key: traverse tree from root to leaf

Key Length and Fanout

- Key length is relevant in B^+ -trees: short keys are good!
 - fanout is maximum number of key-pointer pairs that fit in node
 - long keys result in small fanout
 - small fanout results in more levels



Key Length and Fanout – Example

• Store 40*M* key-pointer pairs in leaf pages (page: 4kB, pointer: 4B)

• 6*B* key: fanout 400 \Rightarrow 3 block reads per accesses

	level	nodes	key-pointer pairs
	1	1	400
	2	400	160,000
	3	160,000	64,000,000
• 96 <i>B</i> key: fanout 40 \Rightarrow 5 block reads per accesses			
	level	nodes	key-pointer pairs
	1	1	40
	2	40	1,600
	3	1,600	64,000
	4	64,000	2,560,000
	5	2,560,000	102,400,000

• 6B key almost twice as fast as 96B key!

Estimate Number of Levels

• Page utilization:

- examples assumes 100% utilization
- typical utilization is 69% (if half-full nodes are merged)

• Number of levels:

 $fanout = \lfloor \frac{node \ size}{key-pointer \ size} \rfloor$ number of levels = $\lceil log_{fanout \times utilization}(leaf \ key-pointer \ pairs) \rceil$

• Previous example with utilization = 69%:

- 6*B* key: fanout = 400, levels = $\lceil 3.11 \rceil = 4$
- 96*B* key: fanout = 40, levels = $\lceil 5.28 \rceil = 6$

Key Compression

• Key compression: produce smaller keys

- reduces number of levels
- adds some CPU cost (ca. 30% per access)
- Key compression is useful if
 - keys are long, for example, string keys
 - data is static (few updates)
 - CPU time is not an issue
- Prefix compression: very popular
 - non-leaf nodes only store prefix of key
 - prefix is long enough to distinguish neighbors
 - example: Cagliari, Casoria, Catanzaro \rightarrow Cag, Cas, Cat

Hash Index



- Hash function:
 - maps keys to integers in range [0..*n*] (hash values)
 - pseudo-randomizing: most keys are uniformly distributed over range
 - similar keys usually have very different hash values!
 - database chooses good hash function for you
- Hash index:
 - hash function is "root node" of index tree
 - hash value is a bucket number
 - bucket either contains records for search key or pointer to overflow chain with records
- Key length:
 - size of hash structure independent of key length
 - key length slightly increases CPU time for hash function
Overflow Chains

- Hash index without overflows: single disk access
- If bucket is full: overflow chain
 - each overflow page requires additional disk access
 - under-utilize hash space to avoid chains!
 - empirical utilization value: 50%
- Hash index with many overflows: reorganize
 - use special reorganize function
 - or simply drop and add index

Bitmap Index

Index for data warehouses

- One bit vector per attribute value (e.g., two for gender)
 - length of each bit vector is number of records
 - bit *i* for vector "male" is set if key value in row *i* is "male"
- Works best if
 - query predicates are on many attributes
 - the individual predicates have weak selectivity (e.g., male/female)
 - all predicates together have strong selectivity (i.e., return few tuples)
- Example: "Find females who have brown hair, blue eyes, wear glasses, are between 50 and 60, work in computer industry, and live in Bolzano"



- *B*⁺-tree index supports
 - point: traverse tree once to find page
 - multi-point: traverse tree once to find page(s)
 - range: traverse tree once to find one interval endpoint and follow pointers between index nodes
 - prefix: traverse tree once to find prefix and follow pointers between index nodes
 - extremal: traverse tree always to left/right (MIN/MAX)
 - ordering: keys ordered by their value
 - grouping: ordered keys save sorting

Which Queries Are Supported?



• Hash index supports

- point: single disk access!
- multi-point: single disk access to first record
- grouping: grouped records have same hash value
- Hash index is useless for
 - range, prefix, extremal, ordering
 - similar key values have dissimilar hash values
 - thus similar keys are in different pages

Experimental Setup

- Employee(<u>ssnum</u>, name, hundreds ...)
- 1,000,000 records
- ssnum is a key (point query)
- hundreds has the same value for 100 employees (multipoint query)
- point query: index on ssnum
- multipoint and range query: index on hundreds
- B^+ -tree and hash indexes are clustered
- bitmap index is never clustered

Index Tuning Data Structures

Experiment: Point Query



Oracle 8i Enterprise Edition on Windows 2000.

- B^+ -tree: search in B+-tree requires additional disk accesses
- Hash index: bucket address is computed without disk access; search key is unique, i.e., bucket overflows are less likely



- Setup: 100 records returned by each query
- B^+ -tree: efficient since records are on consecutive pages
- Hash index: all relevant records in one bucket, but bucket contains also other records; in this experiment, the bucket was too small and an overflow chain was created
- Bitmap index: traverses entire bitmap to fetch a few records

Oracle 8i Enterprise Edition on Windows 2000

DBT – Index Tuning

43 / 78

Index Tuning Data Structures

Experiment: Range Query



• B^+ -tree: efficient since records are on consecutive pages

• Hash index, bitmap index: do not help

Oracle 8i Enterprise Edition on Windows 2000.

DBT – Index Tuning

Outline



- Query Types
- Index Types
- Data Structures

• Composite Indexes

- Indexes and Joins
- Index Tuning Examples

Composite Indexes

- Index on more than one attribute (also "concatenated index")
- Example: Person(ssnum,lastname,firstname,age,address,...)
 - composite index on (lastname,firstname)
 - phone books are organized like that!
- Index can be dense or sparse.
- Dense index on (A, B, C)
 - one pointer is stored per record
 - all pointers to records with the same A value are stored together
 - within one A value, pointers to same B value stored together
 - within one A and B value, pointers to same C value stored together

Composite Indexes – Efficient for Prefix Queries

• Example: composite index on (lastname,firstname) SELECT * FROM Person WHERE lastname='Gates' and firstname LIKE 'Ge%'

• Composite index more efficient than two single-attribute indexes:

- many records may satisfy firstname LIKE 'Ge%'
- condition on lastname and firstname together has stronger selectivity
- two-index solution: results for indexes on lastname and firstname must be intersected
- Dense composite indexes can cover prefix query.

Composite Indexes – Skip Scan in Oracle

- Typically composite index on (lastname,firstname) not useful for SELECT lastname FROM Person WHERE firstname='George'
- Problem: Index covers query, but condition is not a prefix.
- Solution: Index skip scan (implemented in Oracle)
 - composite index on (*A*, *B*)
 - scan each A value until you find required B values
 - then jump to start of next A value
 - partial index scan instead of full table scan!
 - especially useful if A can take few values (e.g., male/female)

Composite Indexes – Multicolumn Uniqueness

- Example: Order(supplier, part, quantity)
 - supplier is not unique
 - part is not unique
 - but (supplier, part) is unique
- Efficient way to ensure uniqueness:
 - create unique, composite index on (supplier, part)
 - CREATE UNIQUE INDEX s_p ON Order(supplier,part)

Composite Indexes – Attribute Order Matters

- Put attribute with more constraints first.
- Example: Geographical Queries
 - table: City(name,longitude,latitude,population)
 SELECT name FROM city
 WHERE population >= 10000 AND latitude = 22
 AND longitude >= 5 AND longitude <= 15

• Efficient: clustered composite index on (latitude,longitude)

- pointers to all result records are packed together
- Inefficient: clustered composite index on (longitude, latitude)
 - each longitude 5 to 15 has some pointers to latitude 22 records
- General geographical queries should use a multi-dimensional index (for example, an R-tree)

Disadvantages of Composite Indexes

• Large key size:

- B^+ tree will have many layers
- key compression can help
- hash index: large keys no problem, but no range and prefix queries supported

• Expensive updates:

- in general, index must be updated when key attribute is updated
- composite index has many key attributes
- update required if any of the attributes is updated

Outline



- Query Types
- Index Types
- Data Structures
- Composite Indexes
- 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⌈log_{M-1}(b/M)⌉ + 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 (10*MB*)

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







DBT – Index Tuning

Outline



- Query Types
- Index Types
- Data Structures
- Composite Indexes
- 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.

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.

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?

Exercise 4 – Clustering Index and Overflows

- Clustering index on Student.ssnum
- Page size: 2*kB*
- Record size in Student table: 1KB (evaluation is a long text)
- Problem: Overflow when new evaluations are added.

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?

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?
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?

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 *SELECT *FROM Employee, StudentFROM Employee, StudentWHERE salary = 12*stipendWHERE salary/12 = stipend
```

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

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 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.

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 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.