Increasing the Performance of analytics applications

MAS DSE 201

SQL as declarative programming

- SQL is a **declarative** programming language:
  - The developer's query only describes **what** result she wants from the database
  - The developer does not describe the algorithm that the database will use in order to compute the result
- The database's optimizer automatically decides what is the most performant algorithm that computes the result of your SQL query
- "Declarative" and "automatic" have been the reason for the success and ubiquitous presence of database systems behind applications
  - Imagine trying to come up yourself with the algorithms that efficiently execute complex queries. Not easy at all.

2

What do you have to do to increase the performance of your db-backed app?

- Does declarative programming mean the developer does not have to think about performance?
  - After all, the database will automatically select the most performant algorithms for the developer's SQL queries
- **No, the developer still has to think and make choices, because…**
  - Size estimation may go wrong
  - Exploration of the space of plans may miss the good one
  - Forgot to create a useful index

3

The developer’s tasks towards increasing performance

1. The developer should select useful database indices
   - Some (non-open source) databases automate the selection of indices, with occasional success.
   - Still, in most cases the developer needs to choose indices.
2. The developer must choose what combination of queries will most efficiently get the data needed by the application
   - Ongoing research at many labs (including UCSD) on how to automate this step.
   - For the foreseeable future, the choice will not be automatic.
3. The developer must diagnose when the database has hugely failed to find a performant algorithm
   - That’s the hard part of advanced analytics programming.
   - Developer may need to reformulate the queries and create temporary results herself.

Diagnostics

- You need to understand few high level points about the performance of your query:

1. Will it benefit from indices? If yes, which are the useful indices?
2. Has the database chosen a hugely suboptimal plan?
3. How can I hack it towards the efficient way?

If all fails…

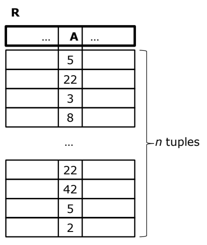
- Precompute some results needed by queries
  - Tedious but effective for aggregations

Boosting performance with indices (a short conceptual summary)

How/when does an index help? Running selection queries without an index

Consider a table R with *n* tuples and the selection query

```
SELECT *
FROM R
WHERE R.A = ?
```

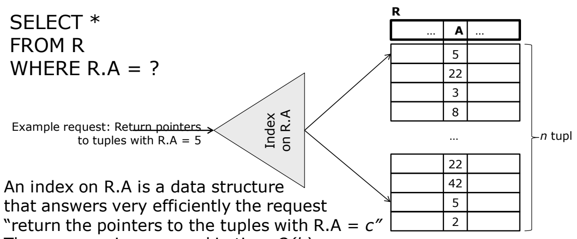


In the absence of an index the *Big-O* cost of evaluating an instance of this query is *O(n)* because the database will need to access the *n* tuples and check the condition R.A = *<provided value>*

How/when does an index help? Running selection queries with an index

Consider a table R with *n* tuples, an index on R.A and assume that R.A has *m* distinct values.
We issue the same query and assume the database uses the index.

```
SELECT *
FROM R
WHERE R.A = ?
```

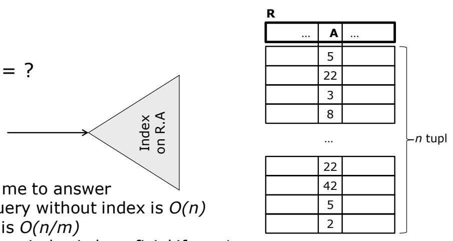


An index on R.A is a data structure that answers very efficiently the request "return the pointers to the tuples with R.A = *c*"
Then a query is answered in time *O(k)* where k is the number of tuples with R.A = c.
Therefore the expected time to answer a selection query is *O(n/m)*

How/when does an index help? Seems good to create an index on R.A assuming m>1

Recall: Table R with *n* tuples, an index on R.A
and assume that R.A has *m* distinct values

```
SELECT *
FROM R
WHERE R.A = ?
```



The expected time to answer
the selection query without index is *O(n)*
and with index is *O(n/m)*
It appears that an index is beneficial if *m>1*
but in practice you will need *m>>1* because **the cost is blocks**!

The mechanics of indices: How to create an index

How to create an index on R.A ?

After you have created table **R**, issue command
CREATE INDEX myIndexOnRA ON R(A)

How to remove the index you previously created ?

DROP INDEX myIndexOnRA

Exercise: Create and then drop an index on
Students.first_name of the enrollment example

After you have created table **students**, issue command
CREATE INDEX students_first_name ON students(first_name)

DROP INDEX students_first_name

Primary keys get an index automatically

Sample relational database

Classes

id	name	number	date_code	start_time	end_time
1	Web stuff	CSE135	TuTh	2:00	3:20
2	Databases	CSE132A	TuTh	3:30	4:50
4	VLSI	CSE121	F	*null*	*null*

Enrollment

id	class	student	credits
1	1	1	4
2	1	2	3
3	4	3	4
4	1	3	3

Students

id	pid	first_name	last_name
1	8888888	John	Smith
2	1111111	Mary	Doe
3	2222222	*null*	Chen

The mechanics of indices: How to use an index in a query

- You do **not** have to change your SQL queries in order to direct the database to use (or not use) the indices you created.
  - All you need to do is to create the index! That's easy…
- The database will decide automatically whether to use (or not use) a created index to answer your query.
- It is possible that you create an index *x* but the database may not use it if it judges that there is a better plan (algorithm) for answering your query, without using the index *x*.

13

The practice of using indices: Experiment, experiment, experiment

- Write your program and queries
- Guess an index *x* that may be useful
- Run your program/query after creating *x*
  - Take note of the "with *x*" response time
- Then run your program/query after dropping *x*
  - Take note of the "without *x*" response time
- If "with x" is better than "without x", then index is probably useful
- There are some pitfalls to take care of…
- The performance difference between two consecutive runs of a program/query may be the result of caching (i.e., RAM caching of disk pages) that happened by the first run and benefited the second run
  - The first run was "cold" and the second run was "warm"

14

3 ways to avoid the cold Vs warm problem

- Reset computer after the first run
  - Bulletproof but takes time to reset for each experiment
- Flush the relevant pages out of the cache by running a query on an irrelevant very large table. Example:
  - Create "irrelevant" table H with integer attribute S. The table H must be larger than the RAM of your system. Then run the query QL = SELECT SUM(S) FROM H
  - The query will fetch all pages of H from disk to RAM and will most likely remove from RAM any pages that were there before
  - Not fully bulletproof: Database buffer managers sometimes do not use "Least Recently Used" strategy. Check if you get the same performance for the same experiment before/after QL
- Run experiments in different databases that have identical data
  - Make many copies of the database. Reset.
  - Each experiment should use another database

15

The practice of choosing indices

- It is easy to perform an experiment that compares two index choices
  - just run your program once for each choice
- In order to avoid experimenting with a ridiculously large number of sets of index choices a little insight on how databases use indices will help

Understanding performance
(preview of next topics)

the condition *m>1* is not enough for making an index useful in practice because…

1. an index on table R is maintained by the database when insert/delete/updates on R
   - Index maintenance costs time
2. obtaining tuple pointers from the index has a run time cost
3. effects of page-based storage in hard disks and solid state drives reduce index usefulness

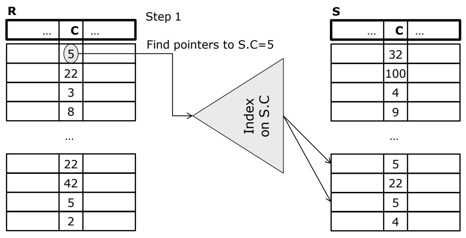
- We discuss next Point 3 and also
- do join and aggregate queries benefit from indices?
- Index interactions

An index may be useful to Join queries

```
SELECT *
FROM R, S
WHERE R.C = S.C
```

```
SELECT *
FROM R JOIN S ON R.C = S.C
```

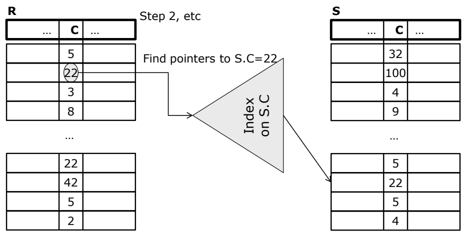
The database may use a plan that uses an index on S.C.
For each R.C it probes index to find matching S tuples



The plan that uses an index on S.C may be the most optimal

```
SELECT *
FROM R, S
WHERE R.C = S.C
```

```
SELECT *
FROM R JOIN S ON R.C = S.C
```



But the index plan has strong competition from the sort-merge plan

The database may execute the join query as follows

- Make a sorted copy of R on R.C
- Make a sorted copy of S on S.C
- Merge the sorted copies outputting pairs (r, s) where r from "sorted R", s from "sorted S" and r.C=s.C
- If R is stored in p_R pages and S is stored in p_S pages then the result is produced in (only!) 3 p_R + 3 p_S page accesses
- Next slides describe how the database does the sorting and merging in so few accesses
- From an application programmer's point of view, it is sufficient to know the 3 p_R + 3 p_S performance result, without the specifics of how the database achieves it

The larger lesson behind sort-merge joins

- In most cases, given a selection/join/aggregation query with tables R1, R2, R3, … and no indices a (decent) database will produce the result by reading/writing at most $k(p_{R1}+p_{R2}+p_{R3}+…)$ pages where *k* is a very small constant (does not depend on size of tables, only depends on query statement)
- that is, the run time should be at most the time it takes to scan the input tables *k* times
- Rule of Thumb 1: If the cost is much higher either
  the database has chosen a horribly wrong plan
  or you deal with a schema and an analytics query where the result does not apply (we will see later how to recognize them – Project 2 is not one of them)
- Rule of Thumb 2: For indexing to be interesting it has to lead to plans that avoid accessing many pages of tables

Recap: Which indices you should consider for selection queries?

- Index when a selection query involves a condition **<attribute> = <value>**
- also when **<attribute> = ?**

No index is useful

Find first names and last names of all students

```
SELECT first_name, last_name
FROM   students;
```

Find all students whose first name is John; project all attributes

index on first name

```
SELECT *
FROM   students
WHERE  first_name = 'John';
```

Find the student whose ID is ?

index on ID, already created

```
SELECT *
FROM   students
WHERE  ID = ?;
```

Find the enrollments of the student whose ID is ?

index on student

```
SELECT *
FROM   enrollment
WHERE  student = ?;
```

Queries with Selections and Joins: Which indices you should try?

```
SELECT  students.pid, students.first_name,
        students.last_name, enrollment.credits
FROM    students, enrollment
WHERE   students.id = enrollment.student
    AND enrollment.class = ? ;
```

Index on Enrollment.class, Students.id (default)

The logic behind choosing promising indices for selection-join queries

- Imagine how the database can use the indices
- For the query of the previous example given the two indices, here is a plan that answers the query very fast in the following steps:

1. Use the enrollment.class index to quickly retrieve the enrollment tuples for the given class
2. For each student id in the retrieved enrollment tuples find the corresponding student record using the students id index

- Cannot be 100% sure that this is the best index choice
  - Eg, it could be that this university offers just 2 or 3 classes and almost all students take each given class. In this case the database will choose a sort-merge plan and, indeed, a sort merge plan is better. The indices are then useless.
- Nevertheless it is worth experimenting with the student.id and enrollment.class index

One selection and two joins: which indices to consider?

Produce a table that shows the pid, first name and last name of every student enrolled in the CSE135 class along with the number of credit units in his/her 135 enrollment

Create indices on Classes.number, Enrollment.class Students.id (default)

```
SELECT  students.pid, students.first_name,
        students.last_name, enrollment.credits
FROM    students, enrollment, classes
WHERE   classes.number = 'CSE135'
        AND students.id = enrollment.student
        AND enrollment.class = classes.id ;
```

Many selections and many joins

produce a table where each row has the name of a 135 student and the name of another class he/she takes

```
SELECT  c_others.name, first_name, last_name
FROM    classes AS c_135, enrollment AS e_135,
        students,
        enrollment AS e_others, classes AS c_others
WHERE   c_135.number = 'CSE135'
        AND c_135.id = e_135.class
        AND e_135.student = students.id
        AND students.id = e_others.student
        AND e_others.class = c_others.id
        AND NOT (c_others.number = 'CSE1
```

Classes.number, Enrollment.class, Students.id, Enrollment.student, Classes.id

Should you use an index in a plain aggregation query? Almost surely No

- Find the average salary in each department that has more than 1 employee:

```
SELECT Dept,AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
HAVING COUNT(Name) >1
```

Should you use an index here? Most likely No

- **Problem:** List all enrolled students and the number of total credits for which they have registered

```
SELECT   students.id, first_name, last_name, SUM(credits)
FROM     students, enrollment
WHERE    students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```

- **Caveat:** In the unlikely case where the vast majority of this university's students are not enrolled in any class (!) the index on students.id becomes useful

Should you consider an index here? Yes

- **Problem:** List all the classes (id's only) in which students of the class "?" are enrolled and also show the number of students (of the class "?") in each one of them. (The "?" is a parameter that will be changed into a class id when a query is executed.)

```
SELECT   e_others.class, COUNT(e.student)
FROM     enrollment e, enrollment e_others
WHERE e.class = ?
 AND e.student = e_others.student
GROUP BY e_others.class
```

enrollment.class, enrollment.student

Exercise: Which indices are needed here?

Sample TPC-H Schema

```
Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
Lineitem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, SName)
```

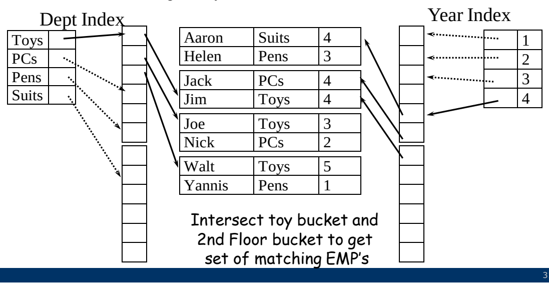
Find the names of suppliers that sell a product that appears in a line item of an order made by a customer who is in Canada

```
SELECT SName
FROM Nation, Customer, Order, LineItem, Product, Supplier
WHERE Nation.NationKey = Cuctomer.NationKey
        AND Customer.CustKey = Order.CustKey
        AND Order.OrderKey=LineItem.OrderKey
        AND LineItem.PartKey= Product.Partkey
        AND Product.Suppkey = Supplier.SuppKey
        AND NName = "Canada"
```

Selection queries with Two conditions: Could both indices be useful simultaneously? Yes: Plans with pointer intersection

Find employees of the Toys dept with 4 years in the company

```
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4
```



The developer's tasks towards increasing performance

1. The developer should select useful database indices
   – See previous notes
2. **Next: The developer must choose what combination of queries will most efficiently get the data needed by the application**
   – Ongoing research at many labs (including UCSD) on how to automate this step.
   – For the foreseeable future, the choice will not be automatic.
3. The developer must diagnose when the database has hugely failed to find a performant algorithm
   – That's the hard part of advanced analytics programming.
   – Developer may need to reformulate the queries and create temporary results herself.

The same app accomplished with different queries

Version 1 (many queries)

```
users = userStatement.executeQuery
        ("SELECT ID, name FROM users") ;
while ( users.next() ) { …
purchases = purchasesStatement.executeQuery(
          "SELECT COUNT(*) AS cnt
         FROM purchases
         WHERE user = " + users.getInt("ID") ) ;
 … }
```

Which version is better for performance?

Version 2 (single query)

```
users_purchases = Statement.executeQuery
                  ("SELECT name, count(*) as cnt
                    FROM users LEFT OUTER JOIN
                                purchases ON (users.id = purchases.user)
                       GROUP BY name, user.ID;") ;
```

A variation: Only the top 50 users alphabetically

Version 1 (many queries)

```
   users = userStatement.executeQuery
            (“SELECT ID, name FROM users ORDER BY name LIMIT 50”) ;
   while ( users.next() ) { …
   purchases = purchasesStatement.executeQuery(
              “SELECT COUNT(*) AS cnt
             FROM purchases
             WHERE user = ” + users.getInt(“ID”) ) ;
    … }
```

Again, which version is better for performance?

Version 2 (single query)

```
     users_purchases = Statement.executeQuery
                        (“SELECT name, count(*) as cnt
                           FROM users LEFT OUTER JOIN
                                         purchases ON (users.id = purchases.user)
                                GROUP BY name, user.ID;
                                ORDER BY name LIMIT 50”) ;
```

Prefer Versions 2

- Versions 2 may be much better than Versions 1
- Versions 2 are at least no worse than Versions 1

Why? Case analysis, assuming index on user.purchases

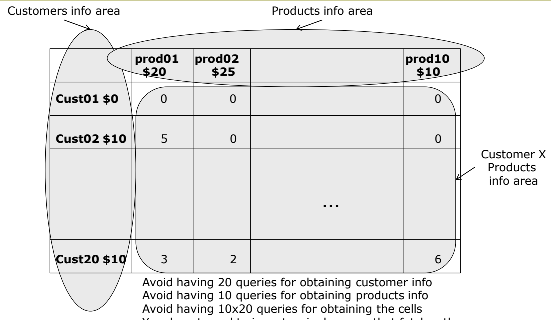
- Case 1: The database runs sort merge for Version 1 queries and sort merge for the Version 2 query
  - Then Version 2 wins big because it scans fewer times
- Case 2: The database runs plan that uses index on purchases.user for Version 1 and runs sort merge for Version 2
  - Then Version 2 is more efficient because the database would have used index-based plan if index-based plan were the more efficient. Notice, Version 1 is misled to use the index because the database receives one-at-a-time individual count queries from the jsp and misses the fact that your jsp activates plenty of count queries!
  - Exception: What if the database chose the wrong plans? The likelihood of this happening is lower than *you* choosing the wrong plan ☺
- Case 3: The database runs plan that uses index for both Versions
  - Then Version 1 and Version 2 perform (almost) the same

Rule of Thumb: Choose few queries because it passes performance choices to the database

Even if you cannot understand the case analysis of previous page, make sure you understand the conclusion:

- Write the smallest number of queries that fetch the exact data that your report presents
  - Mainly avoid the highly-repeated activation of parameterized queries
    - Version 1 repeatedly activates the parameterized count query
  - Do not overdo it about “smallest number of queries”. You need not coalesce qualitatively different areas of a report into the same query in order to achieve a page that is fueled by a single query.

Applying to the problem: 3 qualitatively different areas



	prod01 $20	prod02 $25		prod10 $10
Cust01 $0	0	0		0
Cust02 $10	5	0		0
			…	
Cust20 $10	3	2		6

Avoid having 20 queries for obtaining customer info
Avoid having 10 queries for obtaining products info
Avoid having 10x20 queries for obtaining the cells
You do not need to invent a single query that fetches the data of the 3 areas. It will be too complex and will likely confuse the database

Advanced: Use of temporary tables

- In some challenging report pages your app may benefit from storing the result of a first query into a temp table and then using this temp table in a second query (both first and second query issued during same http request processing)
  - A temp table is eliminated when the connection is closed. Therefore, a solution can be stateless, even if it uses a temp table, since the temp table does not live past the http request processing.
- Open source databases (MySQL and Postgres) often compromise the fundamental benefit from temporary tables by having high costs of creating temp table and writing in them.

The developer's tasks towards increasing performance

1. The developer should select useful database indices
   - Some (non-open source) databases automate the selection of indices, with occasional success.
   - Still, in most cases the developer needs to choose indices.
2. The developer must choose what combination of queries will most efficiently get the data needed by the application
3. The developer must diagnose when the database has hugely failed to find a performant algorithm
   - That's the hard part of advanced analytics programming.
   - Developer may need to reformulate the queries and create temporary results herself.