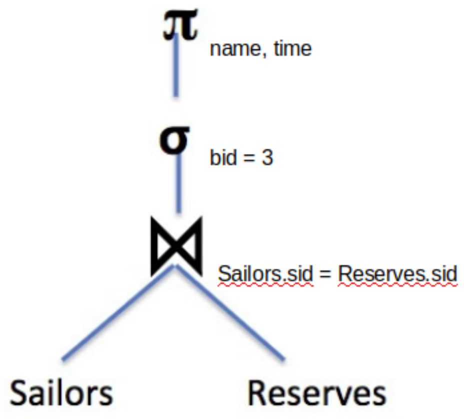
CS W186
Fall 2019

Query Optimization

1 Introduction

When we covered SQL, we gave you a helpful mental model for how queries are executed. First you get all the rows in the FROM clause, then you filter out columns you don't need in the WHERE clause, and so on. This was useful because it guarantees that you will get the correct result for the query, but it is not what databases actually do. Databases can change the order they execute the operations in order to get the best performance. Remember that in this class, we measure performance in terms of the number of IOs. Query Optimization is all about finding the **query plan** that minimizes the number of IOs it takes to execute the query. A query plan is just a sequence of operations that will get us the correct result for a query. We use relational algebra to express it. This is an example of a query plan:



First it joins the two tables together, then it filters out the rows, and finally it projects only the columns it wants. As we'll soon see, we can come up with a much better query plan!

2 Selectivity Estimation

An important property of query optimization is that we have no way of knowing how many IOs a plan will cost until we execute that plan. This has two important implications. The first is that it is impossible for us to guarantee that we will find the optimal query plan - we can only hope to find a good (enough) one using heuristics and estimations. The second is that we need some way to estimate how much a query plan costs. One tool that we will use to estimate a query plan's cost is called **selectivity estimation**. The selectivity of an operator is an approximation for what percentage of pages will make it through the operator onto the operator above it. This is important because if we have an operator that greatly reduces the number of pages that advance to the next stage (like the WHERE clause), we probably want to do that as soon as possible so that the other operators have to work on fewer pages.

Most of the formulas for selectivity estimation are fairly straightforward. For example, to estimate the selectivity of a condition of the form $X = 3$, the formula is 1 / (number of unique values of X). The formulas used in this note are listed below, but please see the lecture/discussion slides for a complete list. In these examples, capital letters are for columns and lowercase letters represent constants.

- **X=a**: 1/(unique vals in X)
- **X=Y**: 1/max(unique vals in X, unique vals in Y)
- **X>a** (max(X) - a) / (max(X) - min(X)
- **cond1 AND cond2**: Selectivity(cond1) * Selectivity(cond2)

3 Selectivity of Joins

Let's say we are trying to join together tables A and B on the condition A.id = B.id. If there was no condition, there would be [A][B] pages in the result, because without the condition, every row from A is joined with every row from B. The join condition is just a condition of the form X=Y, so the selectivity estimation is: 1/max(unique vals for A.id, unique vals for B.id), meaning that the total number of pages we can expect to move on is:

[A][B]/max(unique vals for A.id, unique vals for B.id)

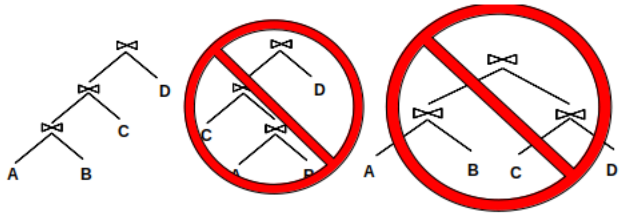
4 Common Heuristics

There are way too many possible query plans for a reasonably complex query to analyze them all. We need some way to cut down the number of plans that we actually consider. For this reason, we will use a few heuristics:

1. Push down projects (π) and selects (σ) as far as they can go
2. Only consider left deep plans
3. Do not consider cross joins unless they are the only option

The first heuristic says that we will push down projects and selects as far as they can go. We touched on why pushing down selects will be beneficial. It reduces the number of pages the other operators have to deal with. But why is pushing projects down beneficial? It turns out this also reduces the number of pages future operators need to deal with. Because projects eliminate columns, the rows become smaller, meaning we can fit more of them on a page, and thus there are fewer pages! Note that you can only project away columns that are not used in the rest of the query (i.e. if they are in a SELECT or a WHERE clause that hasn't yet been evaluated you can't get rid of the column).

The second heuristic says to only consider left deep plans. A left deep plan is a plan where all of the right tables in a join are the base tables (in other words, the right side is never the result of a join itself, it can only be one of the original tables). The following diagram gives some examples of what are and what are not left-deep plans.



Left deep plans are beneficial for two main reasons. First, only considering them greatly reduces the plan space. The plan space is still exponential, but it is a lot smaller than it would be if we considered every plan. Second, these plans can be fully pipelined, meaning that we don't actually have to write the result of a join to disk – we can just pass the pages up one at a time to the next join operator.

The third heuristic is beneficial because cross joins produce a ton of pages which makes the operators above the cross join perform many IOs. We want to avoid that whenever possible.

5 Pass 1 of System R

The query optimizer that we will study in this class is called System R. System R uses all of the heuristics that we mentioned in the previous section. The first pass of System R determines how to access tables optimally or interestingly (we will define interesting in a bit).

We have two options for how to access tables during the first pass:

1. Full Scan
2. Index Scan (for every index the table has built on it)

For both of these scans, we only return a row if it matches all of the single table conditions pertaining to its table because of the first heuristic (push down selects). A condition involving columns from multiple tables is a join condition and cannot yet be applied because we're only considering how to access individual tables. This means the selectivity for each type of scan is the same, because they are applying the same conditions!

The number of IOs required for each type of scan will not be the same, however. For a table

P, a full scan will always take [P] IOs; it needs to read in every page.

For an index scan, the number of IOs depends on how the records are stored and whether or not the index is clustered. Alternative 1 indexes have an IO cost of:

$$\text{(cost to reach level above leaf)} + \text{(num leaves read)}$$

You don't necessarily have to read every leaf, because you can apply the conditions that involve the column the index is built on. This is because the data is in sorted order in the leaves, so you can go straight to the leaf you should start at, and you can scan right until the condition is no longer true.

Example: Important information:

- Table A has [A] pages
- There is an alternative 1 index built on C1 of height 2
- There are 2 conditions in our query: C1 $>$ 5 and C2 $<$ 6
- C1 and C2 both have values in the range 1-10

The selectivity will be 0.25 because both conditions have selectivity 0.5 (from selectivity formulas), and it's an AND clause so we multiply them together. However, we can not use the C2 condition to narrow down what pages we look at in our index because the index is not built on C2. We can use the C1 condition, so we only have to look at 0.5[A] leaf pages. We also have to read the two index pages to find what leaf to start at, for a total number of 2 + 0.5[A] IOs.

For alternative 2/3 indexes, the formula is a little different. The new formula is:

$$\text{(cost to reach level above leaf)} + \text{(num of leaf nodes read)} + \text{(num of data pages read)}.$$

We can apply the selectivity (for conditions on the column that the index is built on) to both the number of leaf nodes read and the number of data pages read. For a clustered index, the number of data pages read is the selectivity multiplied by the total number of data pages. For an unclustered index, however, you have to do an IO for each record, so it is the selectivity multiplied by the total number of records.

Example: Important information:

- Table B with [B] data pages and $|B|$ records
- Alt 2 index on column C1, with a height of 2 and [L] leaf pages

- There are two conditions: C1 > 5 and C2 < 6
- C1 and C2 both have values in the range 1-10

If the index is clustered, the scan will take 2 IOs to reach the index node above the leaf level, it will then have to read 0.5[L] leaf pages, and then 0.5[B] data pages. Therefore, the total is 2 + 0.5[L] + 0.5[B]. If the index is unclustered, the formula is the same except we have to read $0.5|B|$ data pages instead. So the total number of IOs is 2 + 0.5[L] + $0.5|B|$.

The final step of pass 1 is to decide which access plans we will advance to the subsequent passes to be considered. For each table, we will advance the optimal access plan (the one that requires the fewest number of IOs) and any access plans that produce an optimal **interesting order**. An interesting order is when the table is sorted on a column that is either:

- Used in an ORDER BY
- Used in a GROUP BY
- Used in a downstream join (a join that hasn't yet been evaluated. For pass 1, this is all joins).

The first two are hopefully obvious. The last one is valuable because it can allow us to reduce the number of IOs required for a sort merge join later on in the query's execution. A full scan will never produce an interesting order because its output is not sorted. An index scan, however, will produce the output in sorted order on the column that the index is built on. Remember though, that this order is only interesting if it used later in our query!

Example: Let's say we are evaluating the following query:

```
SELECT *
FROM players INNER JOIN teams
ON players.teamid = teams.id
ORDER BY fname;
```

And we have the following potential access patterns:

1. Full Scan players (100 IOs)
2. Index Scan players.age (90 IOs)
3. Index Scan players.teamid (120 IOs)
4. Full Scan teams (300 IOs)
5. Index Scan teams.record (400 IOs)

Patterns 2, 3, and 4 will move on. Patterns 2 and 4 are the optimal pattern for their respective table, and pattern 3 has an interesting order because teamid is used in a downstream join.

6 Passes 2..n

The rest of the passes of the System R algorithm are concerned with joining the tables together. For each pass i, we attempt to join i tables together, using the results from pass i-1 and pass 1. For example, on pass 2 we will attempt to join two tables together, each from pass 1. On pass 5, we will attempt to join a total of 5 tables together. We will get 4 of those tables from pass 4 (which figured out how to join 4 tables together), and we will get the remaining table from pass 1. Notice that this enforces our left-deep plan heuristic. We are always joining a set of joined tables with one base table.

Pass i will produce at least one query plan for all sets of tables of length i that can be joined without a cross join (assuming there is at least one such set). Just like in pass 1, it will advance the optimal plan for each set, and also the optimal plan for each interesting order for each set (if one exists). When attempting to join a set of tables with one table from pass 1, we consider each join the database has implemented. Only one of these joins produces a sorted output - sort merge join, so the only way to have an interesting order is by using sort merge join for the last join in the set. The output of sort merge join will be sorted on the columns in the join condition.

Example: We are trying to execute the following query:

```
SELECT *
FROM A INNER JOIN B
ON A.aid = B.bid
INNER JOIN C
ON b.did = c.cid
ORDER BY c.cid;
```

Which sets of tables will we return query plans for in pass 2? The only sets of tables we will consider on this pass are {A, B} and {B, C}. We do not consider {A, C} because there is no join condition and our heuristics tell us not to consider cross joins. To simplify the problem, let's say we only implemented SMJ and CNLJ in our database and that pass 1 only returned a full table scan for each table. These are the following joins we will consider (the costs are made up for the problem. In practice you will use selectivity estimation and the join cost formulas):

1. A CNLJ B (estimated cost: 1000)
2. B CNLJ A (estimated cost: 1500)
3. A SMJ B (estimated cost: 2000)
4. B CNLJ C (estimated cost: 800)
5. C CNLJ B (estimated cost: 600)
6. C SMJ B (estimated cost: 1000)

Joins 1, 5, and 6 will advance. 1 is the optimal join for the set {A, B}. 5 is optimal for the set {B, C}. 6 is an interesting order because have an ORDER BY clauses that uses c.cid later in the query. We don't advance 3 because A.aid and B.bid are not used after that join so the order isn't interesting.

Now let's go to pass 3. We will consider the following joins (again join costs are made up):

1. Join 1 {A, B} CNLJ C (estimated cost: 10,000)
2. Join 1 {A, B} SMJ C (estimated cost: 12,000)
3. Join 5 {B, C} CNLJ A (estimated cost 8,000)
4. Join 5 {B, C} SMJ A (estimated cost: 20,000)
5. Join 6 {B, C} CNLJ A (estimated cost: 22,000)
6. Join 6 {B, C} SMJ A (estimated cost: 18,000)

Notice that now we can't change the join order, because we are only considering left deep plans, so the base tables must be on the right.

The only plans that will advance now are 2 and 3. 3 is optimal overall for the set of all 3 tables. 2 is optimal for the set of all 3 tables with an interesting order on C.cid (which is still interesting because we haven't evaluated the ORDER BY clause). The reason 2 produces the output sorted on C.cid is that the join condition is B.did = C.cid, so the output will be sorted on both B.did and C.cid (because they are the same). 4 and 6 will not produce output sorted on C.cid because they will be adding A to the set of joined tables so the condition will be A.aid = B.bid. Neither A.aid nor B.bid are used elsewhere in the query, so their ordering isn't interesting to us.