CS 186
Spring 2021

SQL Part 1 - Basic Queries

1 Database Lingo

Relational databases are made up of **tables** (aka relations). A table has a name (we'll call this one `Person`) and looks like this:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

Tables have rows (aka tuples) and columns (aka attributes). In this example, the columns are `name`, `age`, `num_dogs`.

2 Querying a Table

The most fundamental SQL query looks like this:

```
SELECT <columns>
FROM <tbl>;
```

The `FROM` clause tells SQL which table you're interested in, and the `SELECT` clause tells SQL which columns of that table you want to see. For example, consider a table Person(`name`, `age`, `num_dogs`) containing the data below:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

If we executed this SQL query:

```
SELECT name, num_dogs
FROM Person;
```

then we could get the following output.

name	num_dogs
Ace	4
Ada	3
Ben	2
Cho	3

In SQL, however, the order of the rows is nondeterministic unless the query contains an `ORDER BY` (we'll get to this later). So the following output is equally valid:

name	num_dogs
Ben	2
Cho	3
Ace	4
Ada	3

In fact, any ordering of those 4 rows is correct – so unless your query contains an `ORDER BY` clause, don't make any assumptions about the order of your results.

3 Filtering out uninteresting rows

Frequently we are interested in only a subset of the data available to us. That is, even though we might have data about many people or things, we often only want to see the data that we have about very specific people or things. This is where the `WHERE` clause comes in handy; it lets us specify which specific rows of our table we're interested in. Here's the syntax:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>;
```

Once again, let's consider our table `Person(name, age, num_dogs)`. Suppose we want to see how many dogs each person owns — same as before — but this time we only care about the dog-owners who are adults. Let's walk through this SQL query:

```
SELECT name, num_dogs
FROM Person
WHERE age >= 18;
```

When reasoning about query execution you can use the following rule: each clause in a SQL query happens in the order it's written, except for `SELECT` which happens last. This is not necessarily the order the database actually does these operations (more on this later in the semester), but it is easy to think about and will always give us the correct answer. The `FROM` clause tells us we're interested in the `Person` table, so this is the table we start with:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age >= 18`, so we remove the row with Ben, and are left with the following table:

SQL Part 1 - Basic Queries

name	age	num_dogs
Ace	20	4
Ada	18	3
Cho	27	3

And finally, we `SELECT` the columns `name` and `num_dogs` to obtain our final result. (Again, any permutation of this result is equally valid so you shouldn't make any assumptions about the order of the rows.) This gives us our final table:

name	num_dogs
Ace	4
Ada	3
Cho	3

4 Boolean operators

If you want to filter on more complicated predicates, you can use the boolean operators NOT, AND, and OR. For instance, if we only cared about dog-owners who are not only adults, but also own more than 3 dogs, then we would write the following query:

```
SELECT name, num_dogs
FROM Person
WHERE age >= 18
  AND num_dogs > 3;
```

As in Python, this is the order of evaluation for boolean operators:

1. `NOT`
2. `AND`
3. `OR`

That said, it is good practice to avoid ambiguity by adding parentheses even when they are not strictly necessary.

5 Filtering Null Values

In SQL, there is a special value called `NULL`, which can be used as a value for any data type, and represents an "unknown" or "missing" value.

Bear in mind that some values in your database may be `NULL` whether you like it or not, so it's good to know how SQL handles them. It pretty much boils down to the following rules:

- If you do anything with `NULL`, you'll just get `NULL`. For instance if x is `NULL`, then `x > 3`, `1 = x`, and `x + 4` all evaluate to `NULL`. Even `x = NULL` would evaluate to `NULL`; if you want to check whether x is `NULL`, then write `x IS NULL` or `x IS NOT NULL` instead.
- `NULL` is falsey, meaning that `WHERE NULL` is just like `WHERE FALSE`. The row in question does not get included.
- `NULL` short-circuits with boolean operators. That means a boolean expression involving NULL will evaluate to:
  - `TRUE`, if it'd evaluate to `TRUE` regardless of whether the `NULL` value is really `TRUE` or `FALSE`.
  - `FALSE`, if it'd evaluate to `FALSE` regardless of whether the `NULL` value is really `TRUE` or `FALSE`.
  - Or `NULL`, if it depends on the `NULL` value.

Let's walk through this query as an example:

```
SELECT name, num_dogs
FROM Person
WHERE age <= 20
   OR num_dogs = 3;
```

Let's assume we change some values to `NULL`, so after evaluating the `FROM` clause we are left with:

name	age	num_dogs
Ace	20	4
Ada	NULL	3
Ben	NULL	NULL
Cho	27	NULL

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age <= 20` OR `num_dogs = 3`. Let's consider each row one at a time:

- For Ace, `age <= 20` evaluates to `TRUE` so the claim is satisfied.
- For Ada,`age <= 20` evaluates to `NULL` but `num_dogs = 3` evaluates to `TRUE` so the claim is satisfied.
- For Ben, `age <= 20` evaluates to `NULL` and `num_dogs = 3` evaluates to `NULL` so the overall expression is `NULL` which has a falsey value.
- For Cho, `age <= 20` evaluates to `FALSE` and `num_dogs = 3` evaluates to `NULL` so the overall expression evaluates to `NULL` (because it depends on the value of the `NULL`). Because NULL is falsey, this row will be excluded.

Thus we keep only Ace and Ada.

SQL Part 1 - Basic Queries

6 Grouping and Aggregation

When you're working with a very large database, it is useful to be able to summarize your data so that you can better understand the general trends at play. Let's see how.

6.1 Summarizing columns of data

With SQL you are able to summarize entire columns of data using built-in aggregate functions. The most common ones are `SUM`, `AVG`, `MAX`, `MIN`, and `COUNT`. Here are some important characteristics of aggregate functions:

- The input to an aggregate function is the name of a column, and the output is a single value that summarizes all the data within that column.
- Every aggregate ignores NULL values except for `COUNT(*)`. (So `COUNT(<column>)` returns the number of non-NULL values in the specified column, whereas `COUNT(*)` returns the number of rows in the table overall.)

For example, consider this variant of our table `People(name, age, num_dogs)` from earlier, where we are now unsure how many dogs Ben owns:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	NULL
Cho	27	3

With this table in mind ...

- `SUM(age)` is 72.0, and `SUM(num_dogs)` is 10.0.
- `AVG(age)` is 18.0, and `AVG(num_dogs)` is 3.3333333333333333.
- `MAX(age)` is 27, and `MAX(num_dogs)` is 4.
- `MIN(age)` is 7, and `MIN(num_dogs)` is 3.
- `COUNT(age)` is 4, `COUNT(num_dogs)` is 3, and `COUNT(*)` is 4.

So, if we desired the range of ages represented in our database, then we could use the query below and it would produce the result 20. (Technically it would produce a one-by-one table containing the number 20, but SQL treats it the same as the number 20 itself.)

```
SELECT MAX(age) - MIN(age)
FROM Person;
```

Or, if we desired the average number of dogs owned by adults, then we could write this:

```
SELECT AVG(num_dogs)
FROM Person
WHERE age >= 18;
```

6.2 Summarizing Groups of Data

Now you know how to summarize an entire column of your database into a single number. More often than not, though, we want a little finer granularity than that. This is possible with the `GROUP BY` clause, which allows us to split our data into groups and then summarize each group separately. Here's the syntax:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>    -- Filter out rows (before grouping).
GROUP BY <columns>
HAVING <predicate>; -- Filter out groups (after grouping).
```

Notice we also have a brand new `HAVING` clause, which is actually very similar to `WHERE`. The difference?

- `WHERE` occurs *before* grouping. It filters out uninteresting *rows*.
- `HAVING` occurs *after* grouping. It filters out uninteresting *groups*.

To explore all these new mechanics let's see another step-by-step example. This time our query will find the average number of dogs owned, for each adult age represented in our database. We will exclude any age for which we only have one datum.

```
SELECT age, AVG(num_dogs)
FROM Person
WHERE age >= 18
GROUP BY age
HAVING COUNT(*) > 1;
```

Let us assume the `Person` table is now:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3
Ema	20	2
Ian	20	3
Jay	18	5
Mae	33	8
Rex	27	1

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age >= 18`, so we remove the row with Ben.

name	age	num_dogs
Ace	20	4
Ada	18	3
Cho	27	3
Ema	20	2
Ian	20	3
Jay	18	5
Mae	33	8
Rex	27	1

Now for the interesting part. We arrive at the `GROUP BY` clause, which tells us to categorize the data by age. We end up with a group of all the adults 20 years old, a group of all the adults 18 years old, a group of all the adults 27 years old, and a group of all the adults 33 years old. You can now think of the table like this:

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1
Mae	33	8

The `HAVING` clause tells us we only want to keep the groups satisfying the predicate `COUNT(*) > 1` — that is, the groups that contain more than one row. We discard the group that contains only Mae.

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1

Last but not least, every group gets collapsed into a single row. According to our `SELECT` clause, each such row must contain two things:

- The `age` corresponding to the group.
- The `AVG(num_dogs)` for the group.

Our final result looks like this:

age	AVG(num_dogs)
20	3.0
18	4.0
27	2.0

So, to recap, here's how you should go about a query that follows the template above:

- Start with the table specified in the `FROM` clause.
- Filter out uninteresting rows, keeping only the ones that satisfy the `WHERE` clause.
- Put data into groups, according to the `GROUP BY` clause.
- Filter out uninteresting groups, keeping only the ones that satisfy the `HAVING` clause.
- Collapse each group into a single row, containing the fields specified in the `SELECT` clause.

7 A Word of Caution

So that's how grouping and aggregation work, but before we move on I must emphasize one last thing regarding illegal queries. We'll start by considering these two examples:

1. Though it's not immediately obvious, this query actually produces an error:

```
SELECT age, AVG(num_dogs)
FROM Person;
```

What's the issue? `age` is an entire column of numbers, whereas `AVG(num_dogs)` is just a single number. This is problematic because a properly formed table must have the same amount of rows in each column.

2. This query does not work either, for a very similar reason:

```
SELECT age, num_dogs
FROM Person
GROUP BY age;
```

After grouping by age we obtain a table like this:

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1
Mae	33	8

Then the `SELECT` clause's job is to collapse each group into a single row. Each such row must contain two things:

- The `age` corresponding to the group, which is a single number.
- The `num_dogs` for the group, which is an entire column of numbers.

The takeaway from all this? If you're going to do any grouping / aggregation at all, then you must only `SELECT` grouped / aggregated columns.

8 Order By

Before, we mentioned that the ordering of the output rows was usually nondeterministic in SQL. If you want the rows of your table to appear in a certain order you must use an `ORDER BY` clause.

Here is an example query using an `ORDER BY` clause:

```
SELECT name, num_dogs
FROM Person
ORDER BY num_dogs, name;
```

You can include as many columns as you want in the ORDER BY clause. We first sort on the first column listed, and then break any ties with the second column listed, and then break any remaining ties with the third column listed, and so on. By default, the sort order is ascending, but if we want the order in descending order we add the DESC keyword after the column name. If we wanted to sort by the num_dogs ascending and break ties by name descending, we would use the following query:

```
SELECT name, num_dogs
FROM Person
ORDER BY num_dogs, name DESC;
```

9 Limit

Sometimes we only want to see a few rows in our table, even if more rows match all of our other conditions. To do this we can add a `LIMIT` clause to the end of our query to cap the number of rows that will be returned. Note: the same rows may not always be returned by queries using `LIMIT` if an `ORDER BY` is not used or if there are ties in the ordering. Here is a query that only returns one row:

```
SELECT name, num_dogs
FROM Person
LIMIT 1;
```

10 Conclusion

Congratulations - we have covered a lot of SQL! To help remember the ordering of the SQL stages, here is the syntax of a query involving the expressions we have learned so far:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>
GROUP BY <columns>
HAVING <predicate>
ORDER BY <columns>
LIMIT <num>;
```

11 Practice Questions

We have a dogs table that looks like this:

```
CREATE TABLE dogs (
dogid integer,
ownerid integer,
name varchar,
breed varchar,
age integer
)
```

1. Write a query that finds the name of every dog that has an `ownerid=3`.
2. Write a query that lists the 5 oldest dogs' name and age. Break ties by the dog's name in ascending alphabetical order.

3. Write a query that shows how many dogs we have for each breed, but only for breeds that have multiple dogs.

12 Solutions

1.
```
SELECT name
FROM dogs
WHERE ownerid = 3;
```

We're only looking for the name, so that is the only column that goes in the SELECT clause. We are selecting it from the dogs table so the dogs table goes in the FROM clause. We add the only predicate (`ownerid=3`) to the WHERE clause.

2.
```
SELECT name, age
FROM dogs
ORDER BY age DESC, name
LIMIT 5;
```

To find the 5 oldest dogs we will order them by their age (in descending order) and then limit the number of rows returned by our query to 5. The final piece is we need to add a second column to the ORDER BY clause to break ties.

3.
```
SELECT breed, COUNT(*)
FROM dogs
GROUP BY breed
HAVING COUNT(*) > 1;
```

To get statistics for the breeds, we will GROUP BY the breeds column. To only select the groups that have multiple dogs, we add the `COUNT(*) > 1` predicate to the having clause.

Thanks to Sequioa who we have taken a large portion of this note from: https://sequoia-tree.github.io/relational-databases/sql-dml.html

CS 186
Spring 2021

SQL Part 2 - Joins and Subqueries

In our Part 1 note we only looked into querying from one table. Often, however, the data we need to answer a question will be stored in multiple tables. To query from two tables and combine the results we use a **join**.

1 Cross Join

The simplest join is called **cross join**, which is also known as a cross product or a cartesian product. A cross join is the result of combining every row from the left table with every row from the right table. To do a cross join, simply comma separate the tables you would like to join. Here is an example:

```
SELECT *
FROM courses, enrollment;
```

If the courses table looked like this:

num	name
CS186	DB
CS188	AI
CS189	ML

And the enrollment table looked like this:

c_num	students
CS186	700
CS188	800

The result of the query would be:

num	name	c_num	students
CS186	DB	CS186	700
CS186	DB	CS188	800
CS188	AI	CS186	700
CS188	AI	CS188	800
CS189	ML	CS186	700
CS189	ML	CS188	800

The cartesian product often contains much more information than we are actually interested in. Let's say we wanted all of the information about a course (num, name, and num of students enrolled in it). We cannot just blindly join every row from the left table with every row from the right table. There are rows that have two different courses in them! To account for this we will add a **join condition** in the `WHERE` clause to ensure that each row is only about one class.

To get the enrollment information for a course properly we want to make sure that `num` in the courses table is equal to `c_num` in the enrollment table because they are the same thing. The correct query is:

SQL Part 2 - Joins and Subqueries

```
SELECT *
FROM courses, enrollment
WHERE num = c_num;
```

which produces:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

Notice that CS189, which was present in the `courses` table but not in the `enrollment` table, is not included. Since it does not appear as a `c_num` value in enrollment, it cannot fulfill the join condition `num = c_num`.

If we really want CS189 to appear anyway, there are ways to do this that we will discuss later.

2 Inner Join

The cross join works great, but it seems a little sloppy. We are including join logic in the `WHERE` clause. It can be difficult to find what the join condition is. In contrast, the **inner join** allows you to specify the condition in the `ON` clause. Here is the syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1_column_name = table2_column_name;
```

The `table1_column_name = table2_column_name` is the join condition. Let's write the query that gets us all of the course information as an inner join:

```
SELECT *
FROM courses INNER JOIN enrollment
ON num = c_num;
```

This query is logically the exact same query as the query we ran in the previous section. The inner join is essentially syntatic sugar for a cross join with a join condition in the `WHERE` clause like we demonstrated before.

3 Outer Joins

Now lets address the problem we encountered before when we left out CS189 because it did not have any enrollment information. This situation happens frequently. We still want to keep all the data from a relation even if it does not have a "match" in the table we are joining it with. To fix this problem we will use a left outer join. The **left outer join** makes sure that every row from

the left table will appear in the output. If a row does not have any matches with the right table, the row is still included and the columns from the right table are filled in with `NULL`. Let's fix our query:

```
SELECT *
FROM courses LEFT OUTER JOIN enrollment
ON num = c_num;
```

This will produce the following output:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800
CS189	ML	NULL	NULL

Notice that CS189 is now included and the columns that should be from the right table (`c_num`, `students`) are `NULL`.

The **right outer join** is the exact same thing as the left outer join but it keeps all the rows from the right table instead of the left table. The following query is identical to the query above that uses the left outer join:

```
SELECT *
FROM enrollment RIGHT OUTER JOIN courses
ON num = c_num;
```

Notice that I flipped the order of the joins and changed `LEFT` to `RIGHT` because now `courses` is on the right side.

Let's say we add a row to our `enrollment` table now:

c_num	students
CS186	700
CS188	800
CS160	400

But we still want to present all of the information that we have. If we just use a left or a right join we have to pick between using all of the information in the left table or all of the information in the right table. With what we know so far, it is impossible for us to include the information that we have about *both* CS189 and CS160 because they occur in different tables and do not have matches in the other table. To fix this we can use the **full outer join** which guarantees that all rows from each table will appear in the output. If a row from either table does not have a match it will still show up in the output and the columns from the other table in the join will be `NULL`.

To include all of data we have let's change the query to be:

SQL Part 2 - Joins and Subqueries

```
SELECT *
FROM courses FULL OUTER JOIN enrollment
ON num = c_num;
```

which produces the following output:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800
CS189	ML	NULL	NULL
NULL	NULL	CS160	400

4 Name Conflicts

Up to this point our tables have had columns with different names. But what happens if we change the `enrollment` table so that it's `c_num` column is now called `num`?

num	students
CS186	700
CS188	800
CS160	400

Now there is a `num` column in both tables, so simply using `num` in your query is ambiguous. We now have to specify which table's column we are referring to. To do this, we put the table name and a period in front of the column name. Here is an example of doing an inner join of the two tables now:

```
SELECT *
FROM courses INNER JOIN enrollment
ON courses.num = enrollment.num;
```

The result is:

num	name	num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

It can be annoying to type out the entire table name each time we refer to it, so instead we can **alias** the table name. This allows us to rename the table for the rest of the query as something else (usually only a few characters). To do this, after listing the table in the `FROM` we add `AS <alias_name>`. Here is an equivalent query that uses aliases:

```
SELECT *
FROM courses AS c INNER JOIN enrollment AS e
ON c.num = e.num;
```

SQL Part 2 - Joins and Subqueries

num	name	num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

5 Natural Join

Often in relational databases, the columns you want to join on will have the same name. To make it easier to write queries, SQL has the **natural join** which automatically does an equijoin (equijoin = checks if columns are equivalent) on columns with the same name in different tables.

The following query is the same as explicitly doing an inner join on the `num` columns in each table:

```
SELECT *
FROM courses NATURAL JOIN enrollment;
```

The join condition: `courses.num = enrollment.num` is implicit. While this is convenient, natural joins are not often used in practice because they are confusing to read and because adding columns that are not related to the query can change the output.

6 Subqueries

Subqueries allow you to write more powerful queries. Let's look at an example...

Let's say you want to find the course num of every course that has a higher than average num of students. You cannot include an aggregation expression (like `AVG`) in the `WHERE` clause because aggregation happens after rows have been filtered. This may seem challenging at first, but subqueries make it easy:

```
SELECT num
FROM enrollment
WHERE students >= (
    SELECT AVG(students)
    FROM enrollment;
);
```

The output of this query is:

num
CS188
CS186

The inner subquery calculated the average and returned one row. The outer query compared the `students` value for each row to what the subquery returned to determine if the row should be kept.

SQL Part 2 - Joins and Subqueries

Note that this query would be invalid if the subquery returned more than one row because `>=` is meaningless for more than one number. If it returned more than one row we would have to use a set operator like `ALL`.

7 Correlated Subqueries

The subquery can also be correlated with the outer query. Each row essentially gets plugged in to the subquery and then the subquery uses the values of that row. To illustrate this point, let's write a query that returns all of the classes that appear in both tables.

```
SELECT *
FROM classes
WHERE EXISTS (
  SELECT *
  FROM enrollment
  WHERE classes.num = enrollment.num
);
```

As expected, this query returns:

num	name
CS188	AI
CS186	DB

Let's start by examining the subquery. It compares the `classes.num` (the num of the class from the current row) to every `enrollment.num` and returns the row if they match. Therefore, the only rows that will ever be returned are rows with classes that occur in each table. The `EXISTS` keyword is a set operator that returns true if any rows are returned by the subquery and false if otherwise. For `CS186` and `CS188` it will return true (because a row is returned by the subquery), but for `CS189` it will return false. There are a lot of other set operators you should know (including `ANY`, `ALL`, `UNION`, `INTERSECT`, `DIFFERENCE`, `IN`) but we will not cover any others in this note (there is plenty of documentation for these operators online).

8 Subqueries in the From

You can also use subqueries in the `FROM` clause. This lets you create a temporary table to query from. Here is an example:

```
SELECT *
FROM (
    SELECT num
    FROM classes
) as a
WHERE num = 'CS186';
```

Returns:

num
CS186

The subquery returns only the `num` column of the original table, so only the `num` column will appear in the output. One thing to note is that subqueries in the `FROM` cannot usually be correlated with other tables listed in the `FROM`. There is a work around for this, but it is out of scope for this course.

A cleaner way of doing this is using common table expressions (or views if you want to reuse the temporary table in other queries) but we will not cover this in the note.

9 Practice Questions

We will reuse the dogs table from part 1:

```
CREATE TABLE dogs (
 dogid integer,
 ownerid integer,
 name varchar,
 breed varchar,
 age integer,
 PRIMARY KEY (dogid),
 FOREIGN KEY (ownerid) REFERENCES users(userid)
);
```

and add an owners table that looks like this:

```
CREATE TABLE users (
 userid integer,
 name varchar,
 age integer,
 PRIMARY KEY (userid)
);
```

The users own dogs. The `ownerid` column in the dogs table corresponds to the `userid` column of the users table (`ownerid` is a foreign key that references the users table).

1. Write a query that lists the names of all the dogs that "Josh Hug" owns.
2. Write the query above using a different kind of join (i.e. if you used an INNER JOIN, try using a cross join with the join condition in the WHERE).
3. Write a query that finds the name of the user and the number of dogs that user owns for the user that owns the most dogs in the database. Assume that there are no ties (i.e. this query should only return 1 user). Users may share the same name.

4. Now write the same query again, but you can no longer assume that there are no ties.

10 Solutions

1)

```
SELECT dogs.name
FROM dogs INNER JOIN users ON dogs.ownerid = users.userid
WHERE users.name="Josh Hug";
```

We now need information from both tables (the dog name is only in the dogs table and the owner name is only in the users table). The join condition is `dogs.ownerid=users.userid` because we only want to get rows with the dog and its owner in it. Finally we add the predicate to the WHERE clause to only get Josh's dogs.

2)

```
SELECT dogs.name
FROM dogs, users
WHERE dogs.ownerid = users.userid and users.name="Josh Hug";
```

We first do a cross join and then add our join condition to the WHERE clause.

3)

```
SELECT users.name, COUNT(*)
FROM users INNER JOIN dogs on users.userid = dogs.ownerid
GROUP BY users.userid, users.name
ORDER BY COUNT(*) DESC
LIMIT 1;
```

Similarly to question 2 in the part 1 notes, we can use an ORDER BY combined with a LIMIT to select the first n most rows (with n being 1 in this case). We GROUP BY the name because we want our groups to be all about one user. We have to include userid in the GROUP BY, because users may share the same name.

4)

```
SELECT users.name, COUNT(*)
FROM users INNER JOIN dogs ON users.userid = dogs.ownerid
GROUP BY users.userid, users.name
HAVING COUNT(*) >= all(
        SELECT COUNT(*)
        FROM dogs
        GROUP BY ownerid
```

```
)
```

The inner query gets the number of dogs owned by each owner. The owner(s) with the max number of dogs must have a number of dogs that is `>=` all these rows in order to be the max. We put this condition in the HAVING rather than the WHERE clause because it pertains to the groups not the individual rows.

CS 186
Spring 2021

Disk and Files

1 Memory and Disk

Whenever a database processes data, that data must exist in memory. Accessing this data is relatively fast, but once the data becomes very large, it becomes impossible to fit all of it within memory. Disks are used to cheaply store all of a database's data, but they incur a large cost whenever data is accessed or new data is written.

2 Files, Pages, Records

The basic unit of data for relational databases is a **record** (row). These records are organized into **relations** (tables) and can be modified, deleted, searched, or created in memory.

The basic unit of data for disk is a **page**, the smallest unit of transfer from disk to memory and vice versa. In order to represent relational databases in a format compatible with disk, each relation is stored in its own file and its records are organized into pages in the file. Based on the relation's schema and access pattern, the database will determine: (1) type of file used, (2) how pages are organized in the file, (3) how records are organized on each page, (4) and how each record is formatted.

3 Choosing File Types

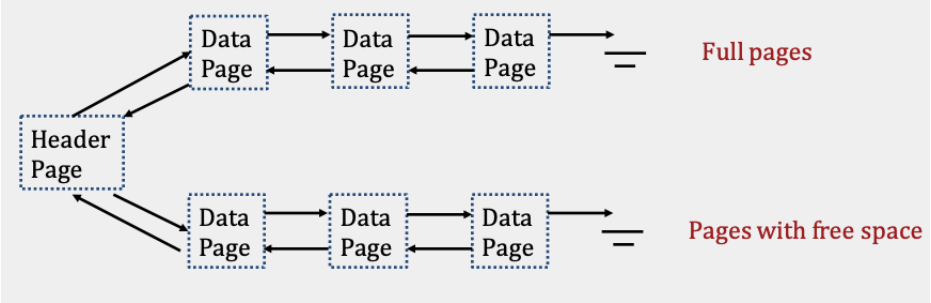
There are two main types of files: Heap Files and Sorted Files. For each relation, the database chooses which file type to use based on the I/O cost of the relation's access pattern. **1 I/O** is equivalent to **1 page read from disk** or **1 page write to disk**, and I/O calculations are made for each file type based on the insert, delete, and scan operations in its access pattern. The file type that incurs less I/O cost is chosen.

4 Heap File

A **heap file is a file type with no particular ordering of pages or of the records on pages** and has two main implementations.

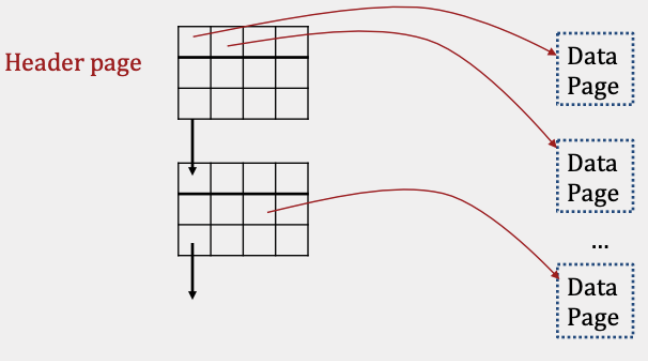
4.1 Linked List Implementation

In the linked list implementation, each data page contains **records**, a **free space tracker**, and **pointers** (byte offsets) to the next and previous page. There is one header page that acts as the start of the file and separates the data pages into full pages and free pages. When space is needed, empty pages are allocated and appended to the free pages portion of the list. When free data pages become full, they are moved from the free space portion to the front of the full pages portion of the linked list. We move it to the front, so we don't have to traverse the entire full pages portion to append it. An alternative is to keep a pointer to the end of this list in the header page. The details of which implementation we use aren't important for this course.



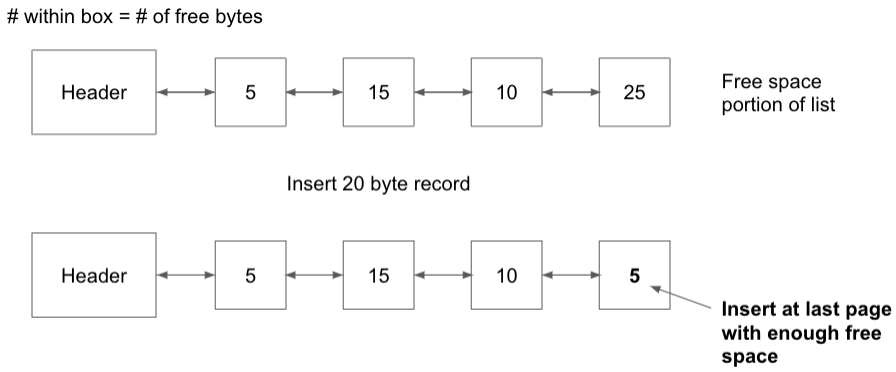
4.2 Page Directory Implementation

The Page Directory implementation differs from the Linked List implementation by only using a **linked list for header pages**. Each header page contains a pointer (byte offset) to the next header page, and its entries contain both a **pointer to a data page** and **the amount of free space left within that data page**. Since our header pages' entries store pointers to each data page, the data pages themselves no longer need to store pointers to neighboring pages.



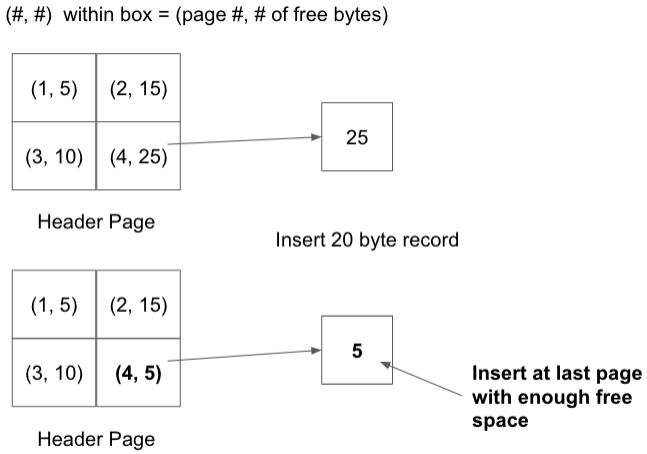
The main advantage of Page Directories over Linked Lists is that inserting records is often faster. To find a page with enough space in the Linked List implementation, the header page and each page in the free portion may need to be read. In contrast, the Page Directory implementation only requires reading at most all of the header pages, as they contain information about how much space is left on each data page in the file.
To highlight this point, consider the following example where a heap file is implemented as both a Linked List and a Page Directory. Each **page is 30 bytes** and a **20 byte record is being inserted** into the file:

Linked List



I/O Cost: 5 (read all pages) + 1 (write last data page) = **6 I/Os**

Page Directory



I/O Cost: 1 (read header) + 1 (read data) + 1 (write data) + 1 (write header) = **4 I/Os**

This is only a small example and as the number of pages increases, a scenario like this would cause insertion in a linked list to be much more expensive than insertion in a page directory.

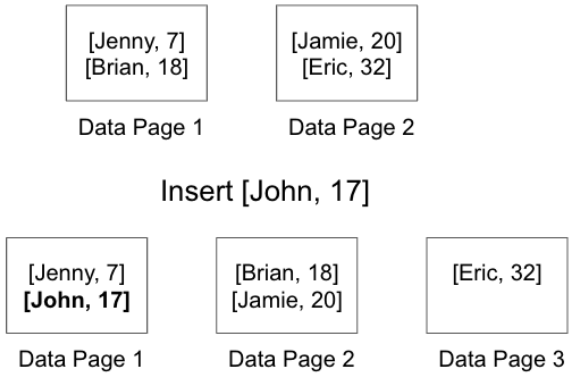
Regardless of the implementation used, heap files provide faster insertions than sorted files (discussed below) because records can be added to any page with free space, and finding a page with enough free space is often very cheap. However, searching for records within heap files requires a full scan every time. Every record on every page must be looked at because records are unordered, resulting in a linear cost of N I/Os for every search operation. We will see that sorted files are much better at searching for recordings.

5 Sorted Files

A **sorted file is a file type where pages are ordered and records within each page are sorted by key(s)**.

These files are implemented using Page Directories and enforce an ordering upon data pages based on how records are sorted. Searching through sorted files takes logN I/Os where N = # of pages since binary search can be used to find the page containing the record. Meanwhile, insertion, in the average case, takes logN + N I/Os since binary search is needed to find the page to write to and that inserted record could potentially cause all later records to be pushed back by one. On average, N / 2 pages will need to be pushed back, and this involves a read and a write IO for each of those pages, which results in the N I/Os term.

In the example below, each data page can store upto 2 records, so inserting a record in Data Page 1, requires a read and a write of all pages that follow, since the rest of the records need to be pushed back.

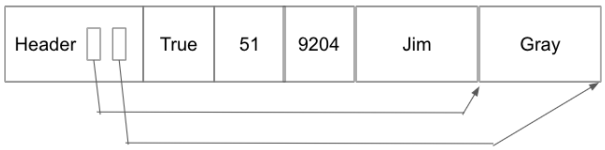


6 A Note on Counting Header Pages

An area of confusion that often arises while counting the number of I/Os that an operation costs, is whether or not to consider the cost of accessing the header pages in the file. For all problems in this course that only specify whether a file is a heap file or sorted file without providing the underlying implementation, you should ignore the I/O cost associated with reading/writing the file's header pages. However, for all problems that provide a specific file implementation (i.e. heap file implemented with a linked list or page directory), you must include the I/O cost associated with reading/writing the file's header pages.

7 Record Types

Record types are completely determined by the relation's schema and come in 2 types: **Fixed Length Records** (FLR) and **Variable Length Records** (VLR). FLRs only contain fixed length fields (integer, boolean, date, etc.), and FLRs with the same schema consist of the same number of bytes. Meanwhile, VLRs contain both fixed length and variable length fields (eg. varchar), resulting in each VLR of the same schema having a potentially different number of bytes. VLRs store all fixed length fields before variable length fields and use a record header that contains pointers to the end of the variable length fields.



Regardless of the format, every record can be uniquely identified by its record id - [**page #, record # on page**].

8 Page Formats

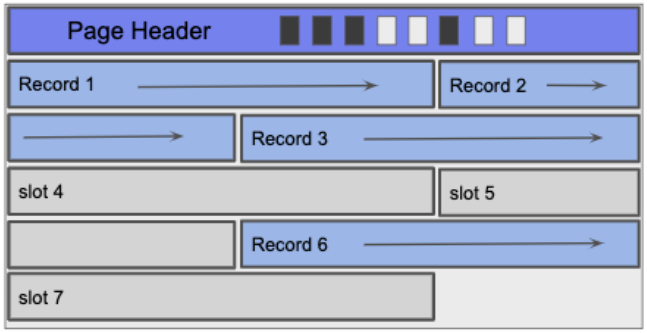
8.1 Pages with Fixed Length Records

Pages containing FLRs always use page headers to store the number of records currently on the page.

If the page is packed, there are no gaps between records. This makes insertion easy as we can calculate the next available position within the page using the # of existing records and the length

of each record. Once this value is calculated, we insert the record at the computed offset. Deletion is slightly trickier as it requires moving all records after the deleted record towards the top of the page by one position to keep the page packed.

If the page is unpacked, the page header typically stores an additional **bitmap** that breaks the page into slots and tracks which slots are open or taken.



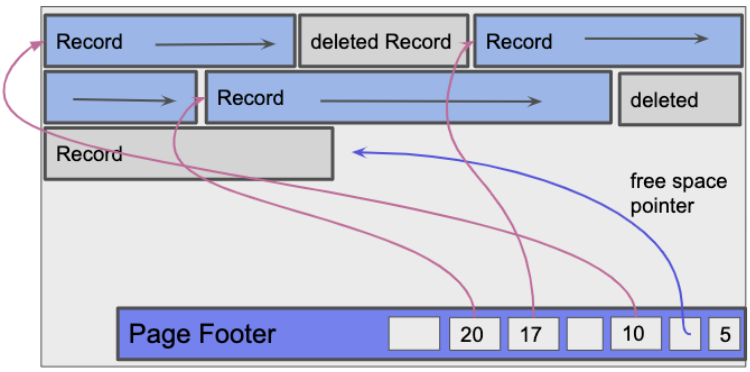
Using the bitmap, insertion involves finding the first open bit, setting the new record in the corresponding slot, and then setting the bit for that slot. With deletion, we clear the deleted record's corresponding bit so that future inserts can overwrite that slot.

8.2 Pages with Variable Length Records

The main difference between variable length records and fixed length records is that we no longer have a guarantee on the size of each record. To work around this, each page uses a **page footer** that maintains a **slot directory** tracking **slot count**, **a free space pointer**, and **entries**. The footer starts from the bottom of the page rather than the top so that the slot directory has room to grow when records are inserted.

The slot count tracks the total number of slots. This includes both filled and empty slots. The free space pointer points to the next free position within the page. Each entry in the slot directory consists of a [**record pointer**, **record length**] pair.

If the page is unpacked, deletion involves finding the record's entry within the slot directory and setting both the record pointer and record length to null.



For future insertions, the record is inserted into the page at the free space pointer and a new [pointer, length] pair is set in any available null entry. In the case where there are no null entries, a new entry is added to the slot directory for that record. The slot count is used to determine which offset the new slot entry should be added at, and then the slot count is incremented. Periodically, records will be reorganized into a packed state where deleted records are removed to make space for future insertions.

If the page is packed, deletion involves removing the record's entry within the slot directory. Additionally, records after the deleted record must be moved towards the top of the page by one position and the corresponding slot directory entries shifted towards the bottom of the page by one position. For insertion, the record is inserted at the free space pointer and a new entry is added every time if all slots are full.

9 Practice Questions

1. Given a heap file implemented as a Page Directory, what is the I/O cost to insert a record in the worst case? The directory contains 4 header pages and 3 data pages for each header page. Assume that at least one data page has enough space to fit the record.

2. What is the smallest size, in bytes, of a record from the following schema? Assume that the record header is 5 bytes. (boolean = 1 byte, date = 8 bytes)

   name VARCHAR
   student BOOLEAN
   birthday DATE
   state VARCHAR

3. 4 VLRs are inserted into an empty page. What is the size of the slot directory? (int = 4 bytes) Assume there are initially no slots in the slot directory.

4. Assume you have a heap file implemented in a linked list structure with the header page connected to a linked list of full pages, and connected to a linked list of pages with free space. There are 10 full pages and 5 pages with free space. In the worst case, how many pages would you have to read to see if there is a page with enough space to store some record with a given size?

10 Solutions

1. In the worst case, the only data page with enough free space is on the very last header page. Therefore, the cost is 7 I/Os.

   4 (read header pages) + 1 (read data) + 1 (write data) + 1 (write last header) = 7

2. The smallest size of the VLR is 14 bytes and occurs when both name and state are null.

   5 (record header) + 1 (boolean) + 8 (date) = 14

3. The slot directory contains a slot count, free space pointer, and entries, which are record pointer, record size pairs. Since pointers are just byte offsets within the page, the size of the directory is 40 bytes.

   4 (slot count) + 4 (free space) + (4 (record pointer) + 4 (record size)) * 4 (# records) = 40

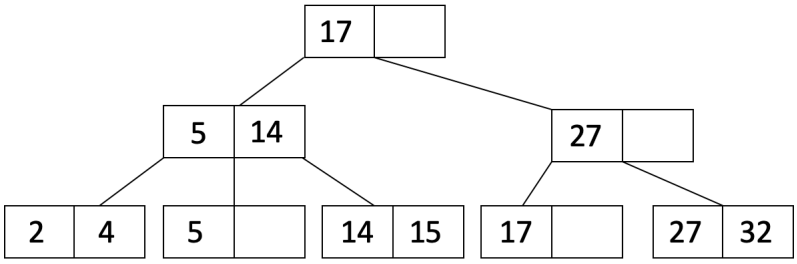
4. 6 - You have to read the header page and then you may have to sequentially read all 5 pages with free space to see if any of them have enough space for this record, for a total of 6 pages.

CS 186
Spring 2021

B+ Trees

1 Introduction

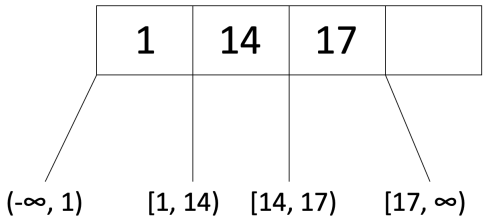
An index is a data structure that helps speed up reads on a specific key. We use indexes to make queries run faster, especially those that are run frequently. Consider a web application that looks up the record for a user in a Users table based on the username during the login process. An index on the username column will make login faster by quickly finding the row of the user trying to log in. In this course note, we will learn about B+ trees, which is a specific type of index. Here is an example of what a B+ tree looks like:



2 Properties

- The number d is the order of a B+ tree. Each node (with the exception of the root node) must have $d \leq x \leq 2d$ entries assuming no deletes happen (it's possible for leaf nodes to end up with $<$ d entries if you delete data). The entries within each node must be **sorted**.
- In between each entry of an inner node, there is a pointer to a child node. Since there are at most $2d$ entries in a node, inner nodes may have at most $2d + 1$ child pointers. This is also called the tree's fanout.
- The keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.

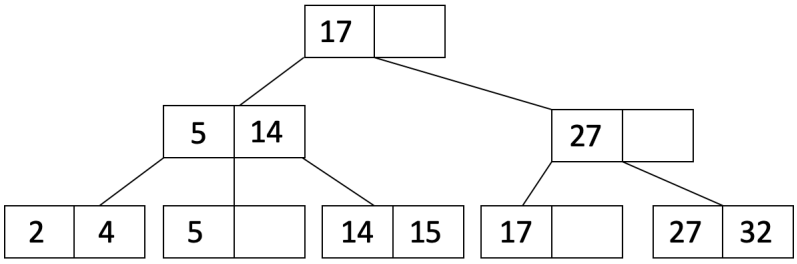
For example, here is a node of an order $d = 2$ tree:



Note that the node satisfies the order requirement ($d \leq x \leq 2d$) because $d = 2$ and this node has 3 entries which satisfies $2 \leq x \leq 4$.

- Because of the sorted and children property, we can traverse the tree down to the leaf to find our desired record. This is similar to BSTs (Binary Search Trees).
- Every root to leaf path has the same number of **edges** - this is the height of the tree. In this sense, B+ trees are **always** balanced. In other words, a B+ tree with just the root node has height 0.
- Only the leaf nodes contain records (or pointers to records - this will be explained later). The inner nodes (which are the non-leaf nodes) do not contain the actual records.

For example, here is an order $d = 1$ tree:



3 Insertion

To insert an entry into the B+ tree, follow this procedure:

(1) Find the leaf node L in which you will insert your value. You can do this by traversing down the tree. Add the key and the record to the leaf node in order.

(2) If L overflows (L has more than $2d$ entries)...

  (a) Split into L_1 and L_2. Keep d entries in L_1 (this means $d + 1$ entries will go in L_2).

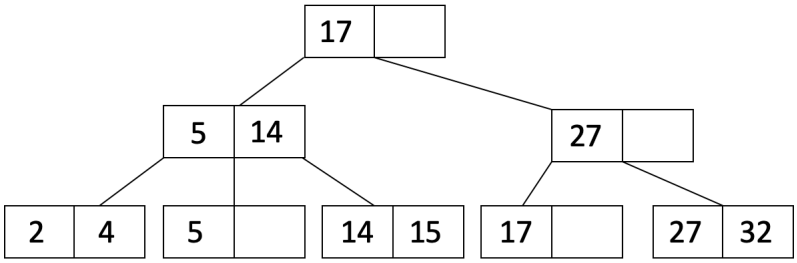
  (b) If L was a leaf node, **COPY** L_2's first entry into the parent. If L was not a leaf node, **MOVE** L_2's first entry into the parent.

  (c) Adjust pointers.

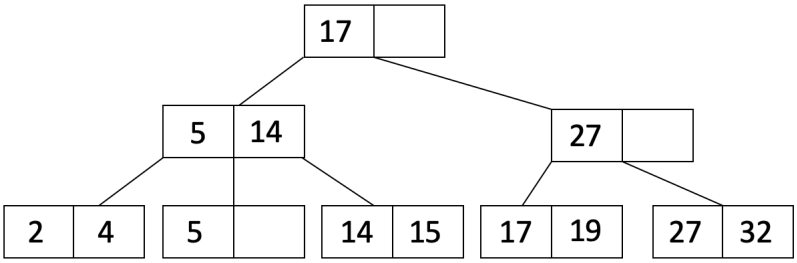
(3) If the parent overflows, then recurse on it by doing step 2 on the parent.

Note: we want to **COPY** leaf node data into the parent so that we don't lose the data in the leaf node. Remember that every key that is in the table that the index is built on must be in the leaf nodes! Being in a inner node does not mean that key is actually still in the table. On the other hand, we can **MOVE** inner node data into parent nodes because the inner node does not contain the actual data, they are just a reference of which way to search when traversing the tree.

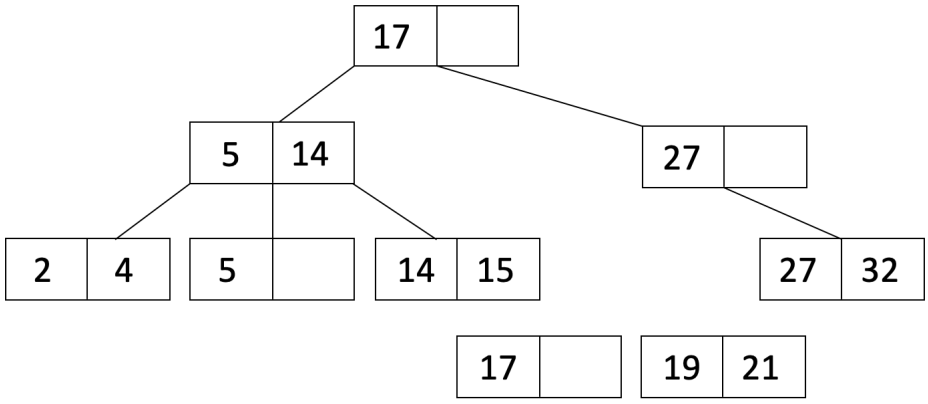
Let's take a look at an example to better understand this procedure! We start with the following order $d = 1$ tree:



Let's insert 19 into our tree. When we insert 19, we see that there is space in the leaf node with 17:

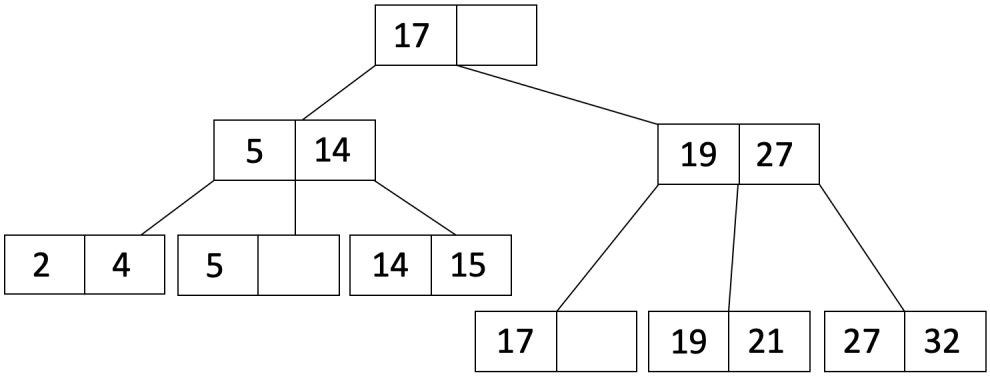


Now let's insert 21 into our tree. When we insert 21, it causes one of the leaf nodes to overflow. Therefore, we split this leaf node into two leaf nodes as shown below:

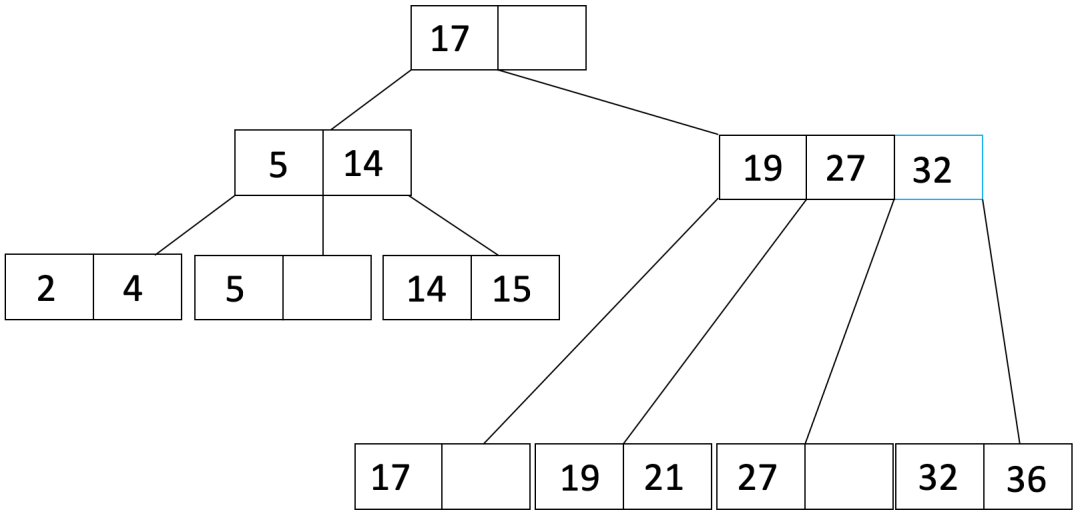


L_1 is the leaf node with 17, and L_2 is the leaf node with 19 and 21.

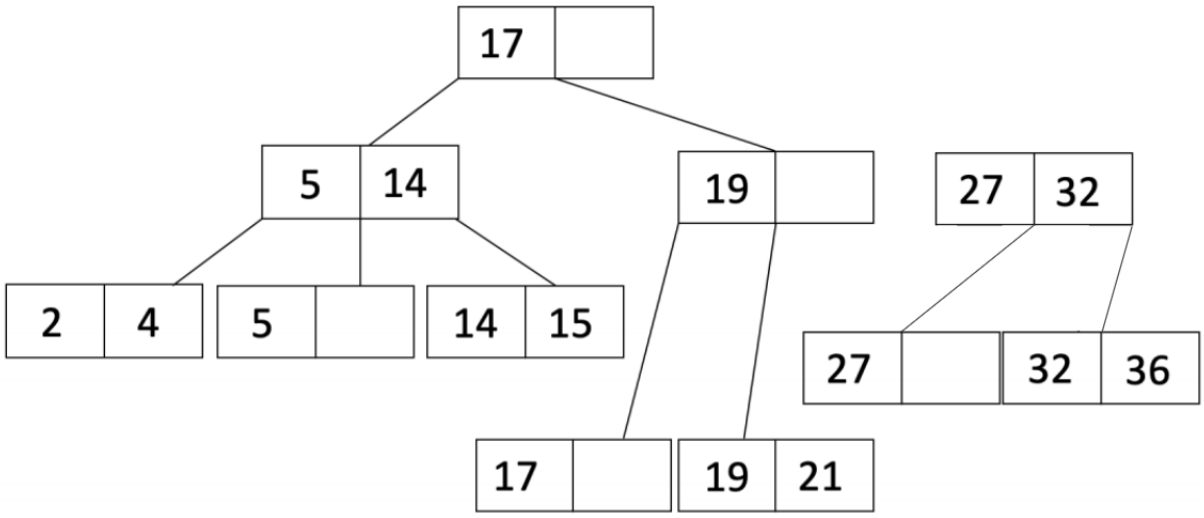
Since we split a leaf node, we will **COPY** L_2's first entry up to the parent and adjust pointers. We also sort the entries of the parent to get:



Let's do one more insertion. This time we will insert 36. When we insert 36, the leaf overflows so we will do the same procedure as when we inserted 21 to get:

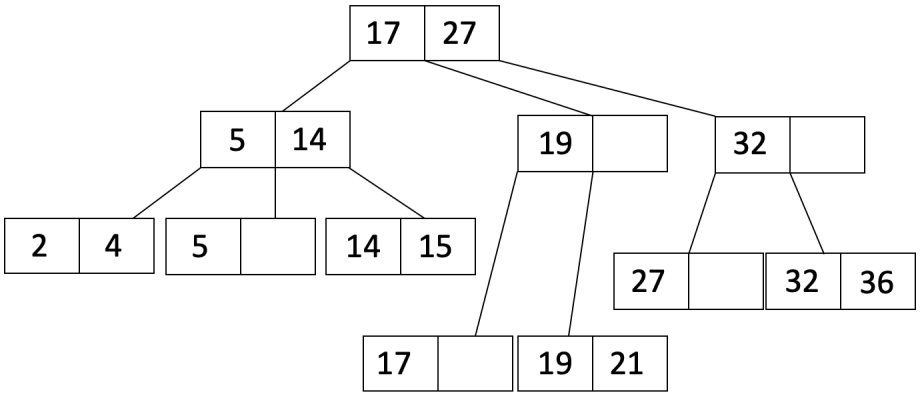


Notice that now the parent node overflowed, so now we must recurse. We will split the parent node to get:



L_1 is the inner node with 19, and L_2 is inner the node with 27 and 32.

Since it was an inner node that overflowed, we will **MOVE** L_2's first entry up to the parent and adjust pointers to get:



4 Deletion

To delete a value, just find the appropriate leaf and delete the unwanted value from that leaf. That's all there is to it. (Yes, technically we could end up violating some of the invariants of a B+ tree. That's okay because in practice we get *way* more insertions than deletions so something will quickly replace whatever we delete.)

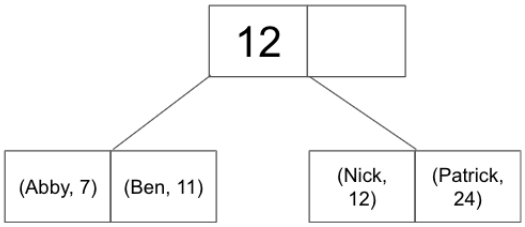
Reminder: We never delete inner node keys because they are only there for search and not to hold data.

5 Storing Records

Up until now, we have not discussed how the records are actually stored in the leaves. Let's take a look at that now. There are three ways of storing records in leaves:

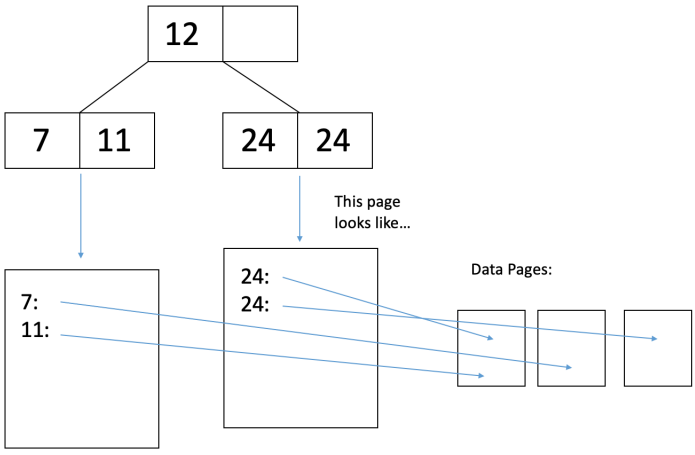
- **Alternative 1**

  In the Alternative 1 scheme, the leaf pages are the data pages. Rather than containing pointers to records, the leaf pages contain the records themselves.



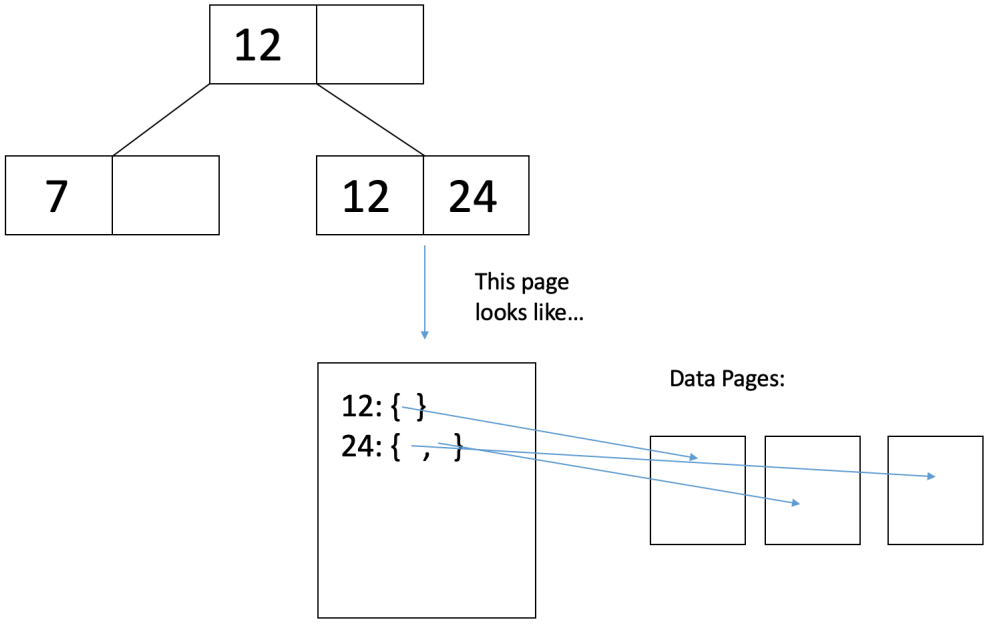
- **Alternative 2**

  In the Alternative 2 scheme, the leaf pages hold pointers to the corresponding records.



- **Alternative 3**

  In the Alternative 3 scheme, the leaf pages hold lists of pointers to the corresponding records. This is more compact than Alternative 2 when there are multiple records with the same leaf node entry.

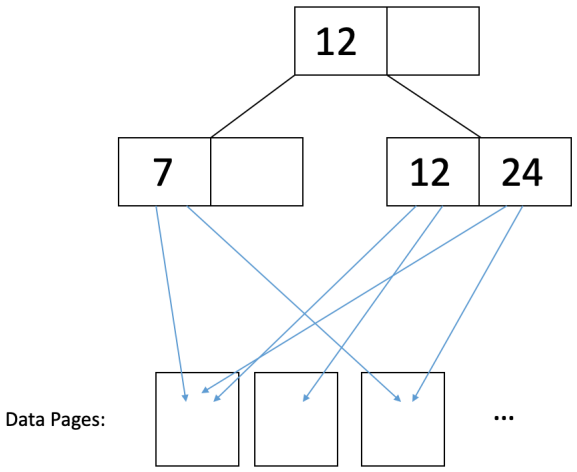


6 Clustering

Now that we've discussed how records are stored in the leaf nodes, we will also discuss how data on the data pages are organized. Clustered/unclustered refers to how the data pages are structured. Unclustering only applies to Alternative 2 or 3.

- **Unclustered**

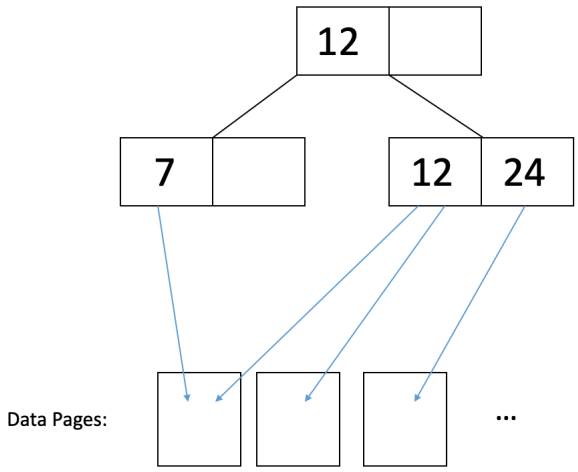
  In an unclustered index, the data pages are complete chaos. Thus, odds are that you're going to have to read a separate page for each of the records you want. For instance, consider this illustration:



  In the figure above, if we want to read records with 12 and 24, then we would have to read in each of the data pages they point to in order to retrieve all the records associated with these keys.

- **Clustered**

  In a clustered index, the data pages are sorted on the same index on which you've built your B+ tree. This does not mean that the data pages are sorted exactly, just that keys are roughly in the same order as data. The difference in I/O cost therefore comes from caching, where two records with close keys will likely be in the same page, so the second one can be read from the cached page. Thus, you typically just need to read one page to get all the records that have a common / similar key. For instance, consider this illustration:



  In the figure above, we can read records with 7 and 12 by reading 2 pages. If we do sequential reads of the leaf node values, the data page is largely the same. In conclusion,

- UNCLUSTERED = $\sim$ 1 I/O per record.
- CLUSTERED = $\sim$ 1 I/O per page of records.

7 Counting IO's

Here's the general procedure. It's a good thing to write on your cheat sheet:

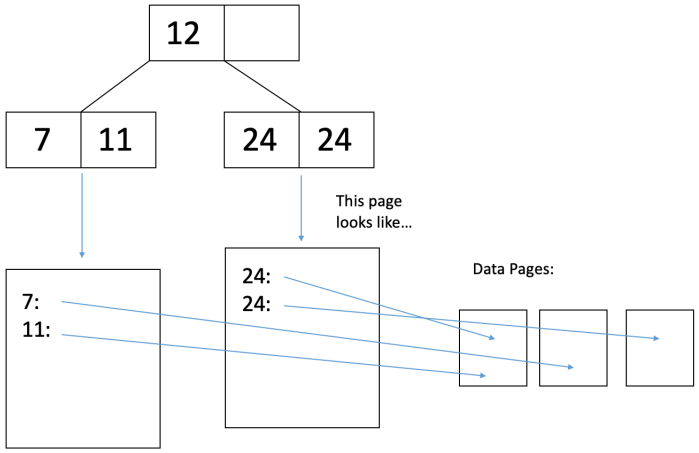
(1) Read the appropriate root-to-leaf path.

(2) Read the appropriate data page(s). If we need to read multiple pages, we will allot a read IO for each page. In addition, we account for clustering for Alt. 2 or 3 (see below.)

(3) Write data page, if you want to modify it. Again, if we want to do a write that spans multiple data pages, we will need to allot a write IO for each page.

(4) Update index page(s).

Let's look at an example. See the following **Alternative 2 Unclustered** B+ tree:



We want to delete the only 11-year-old from our database. How many I/Os will it take?

- One I/O for each of the 2 relevant index pages (an index page is an inner node or a leaf node).
- One I/O to read the data page where the 11-year-old's record is. Once it's in memory we can delete the record from the page.
- One I/O to write the modified data page back to disk.
- Now that there are no more 11-year-olds in our database we should remove the key "11" from the leaf page of our B+ tree, which we already read in Step 1. We do so, and then it takes one I/O to write the modified leaf page to disk.
- Thus, the total cost to delete the record was 5 I/Os.

8 Bulk Loading

The insertion procedure we discussed before is great for making additions to an existing B+ tree. If we want to construct a B+ from scratch, however, we can do better. This is because if we use the insertion procedure we would have to traverse the tree each time we want to insert something new. Instead, we will use **bulkloading**:

(1) Sort the data on the key the index will be built on.

(2) Fill leaf pages until some fill factor f.

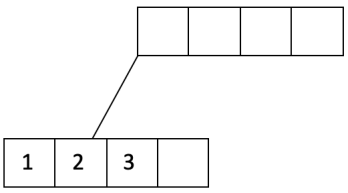
(3) Add a pointer from parent to leaf page. If the parent overflows, we will follow a procedure similar to insertion. We will split the parent into two nodes:

   (a) Keep d entries in L_1 (this means $d+1$ entries will go in L_2).

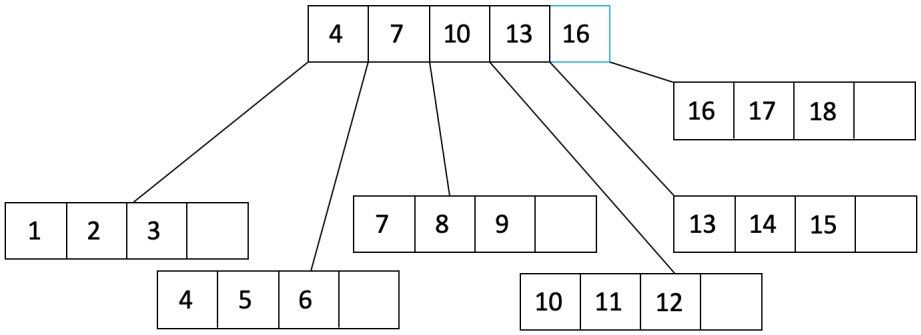
   (b) Since a parent node overflowed, we will **MOVE** L_2's first entry into the parent.

(4) Adjust pointers.

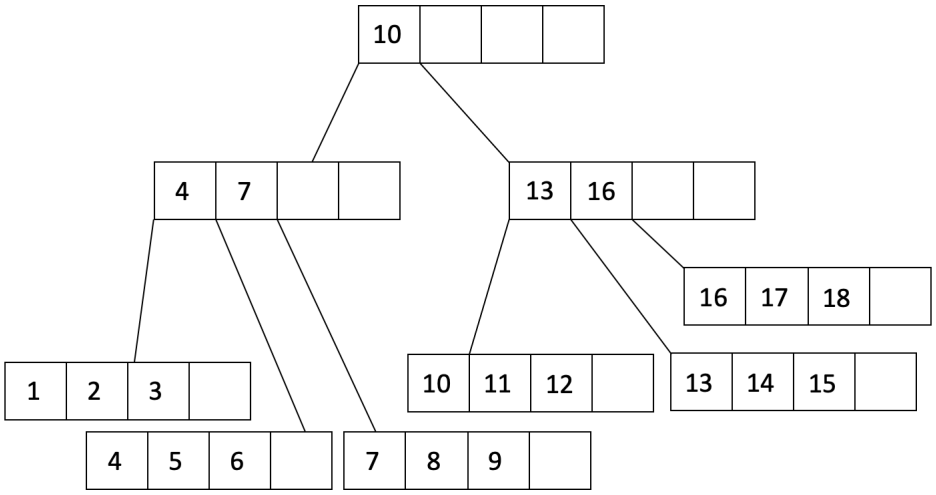
Let's look at an example. Let's say our fill factor is $\frac{3}{4}$ and we want to insert $1, \ldots, 20$ into an order $d = 2$ tree. We will start by filling a leaf page until our fill factor:



We have filled a leaf node to the fill factor of $\frac{3}{4}$ and added a pointer from the parent node to the leaf node. Let's continue filling:

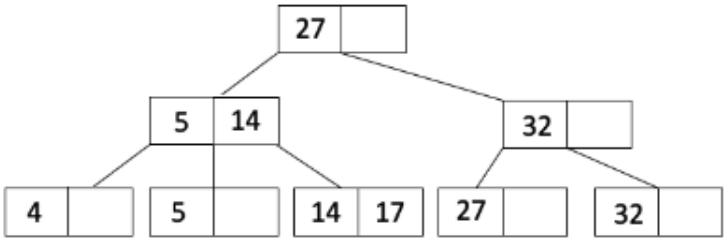


In the figure above, we see that the parent node has overflowed. We will split the parent node into two nodes and create a new parent:



9 Practice Questions

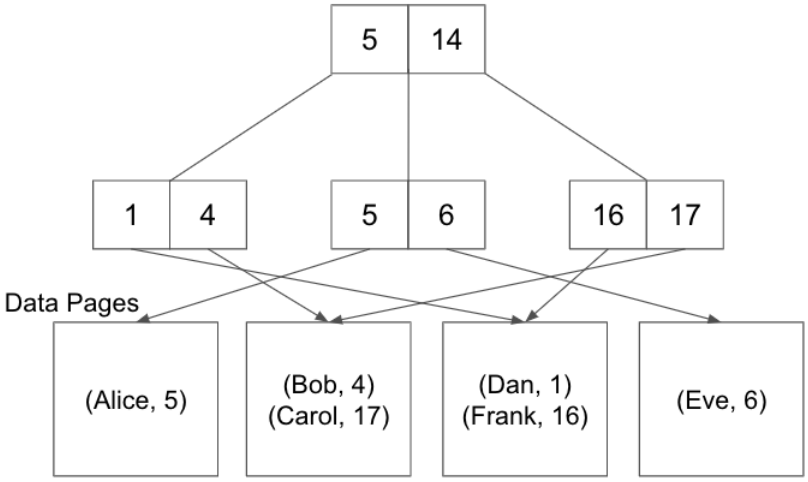
1. What is the maximum number of entries we can add to the following B+ tree without increasing its height?



2. What is the minimum number of entries we can add to the B+ tree above which will cause the height to increase?

3. How many I/Os would it cost to insert an entry into our table if we had a height 2, unclustered alternative 3 B+ tree in the worst case? Assume that the cache is empty at the beginning and there are at most 10 entries with the same key. Assume each page is at most $\frac{2}{3}$ full.

4. Assume we have the following Alternative 2 unclustered B+ tree built over the age field of a relation. How many I/Os will it take to find all records where age ≥ 6?



10 Solutions

1. The maximum number of entries we can add without increasing the height is the total capacity of a height 2, order $d = 1$ tree minus the current number of entries. The total capacity is $(2d)(2d+1)^h = (2)(3^2) = 18$. The current number of entries is 6 because the entries are in the leaf nodes. Therefore, we can add a maximum of $18 - 6 = 12$.

2. The minimum number of entries we can add to cause the height to increase will be 3.

   We will add 20 to the fullest leaf node which is $14, 17$ which will cause it to split into 14 and $17, 20$. Then we copy 17 to its parent $5, 14$, which causes it to split into 5 and 17 and 14 will be pushed up so the root will have $14, 27$.

   Then we will insert 21 into the fullest leaf which will now be $17, 20$, which will cause it to split into 17 and $20, 21$. Then we copy 20 up to its parent which will become $17, 20$.

   Then we can insert 22 into the fullest leaf $20, 21$ which will split into 20 and $21, 22$. Then we copy 21 up to the parent $17, 20$, which will cause it to split into 17 and 21 because 20 will be pushed up to its parent (the root). This root will also split because it already has $14, 27$. When the root splits, we know that the height has increased.

3. First, we need to check if our table already contains this entry. We will search down the B+ tree for the key which will cost us 3 I/Os because the tree is height 2. Then we get to a leaf node and we see that there are at most 10 entries with the same key. We need to check each of these entries to make sure it's not the same as our current entry which is 10 I/Os because our B+ tree is unclustered. In the worst case, none of the 10 entries match the entry we want to insert so we will have to go ahead and add it to the table. We will add the entry to any page that has space (we already have this page in our cache from the previous part so this does not incur an I/O) and then writing this page back to disk which will cost 1 I/O. We also have to update our B+ tree leaf node to include a pointer to this new entry so we will have to write the node back to disk which is 1 I/O. Therefore, we will have a total of $3 + 10 + 1 + 1 = 15$ I/Os.

4. 6 I/Os. The general idea is that we need to look up which leaf node corresponds to age = 6 and scan through all leaf nodes to the right. Since this is an Alternative 2 B+ tree, for each leaf node we will also need to read in the data pages the records lie on (1 I/O per matching record since the index is unclustered).

   We will incur:

   - 1 I/O to read in the root node: 5, 14
   - 1 I/O to read in the 5, 6 leaf node
   - 1 I/O to read in the 4th data page, which contains (Eve, 6)
   - 1 I/O to read in the 16, 17 leaf node

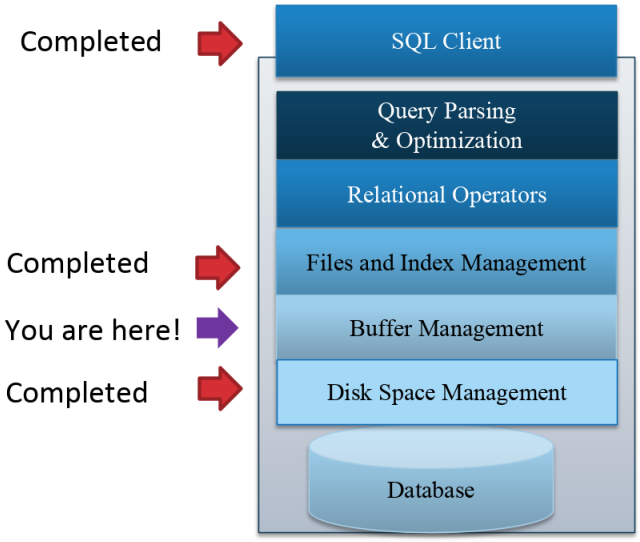
- 1 I/O to read in the 3rd data page, which contains (Frank, 16)
- 1 I/O to read in the 2nd data page, which contains (Carol, 17)

CS 186
Spring 2021

Buffer Management

1 Introduction

So far, we have discussed how disk space is managed at the lowest level of the database management system and how files and indexes are managed in our page-based database system. We will now explore the interface between these two levels on the DBMS - the buffer manager.



The buffer manager is responsible for managing pages in memory and processing page requests from the file and index manager. Remember, space on memory is limited, so we cannot afford to store all pages in the buffer pool. The buffer manager is responsible for the eviction policy, or choosing which pages to evict when space is filled up. When pages are evicted from memory or new pages are read in to memory, the buffer manager communicates with the disk space manager to perform the required disk operations.

2 Buffer Pool

Memory is converted into a buffer pool by partitioning the space into frames that pages can be placed in. A buffer frame can hold the same amount of data as a page can (so a page fits perfectly into a frame). To efficiently track frames, the buffer manager allocates additional space in memory for a metadata table.

Frame ID	Page ID	Dirty Bit	Pin Count
0	5	1	3
1	3	0	1
2	10	1	0
3			

The table tracks 4 pieces of information:

1. **Frame ID** that is uniquely associated with a memory address
2. **Page ID** for determining which page a frame currently contains
3. **Dirty Bit** for verifying whether or not a page has been modified
4. **Pin Count** for tracking the number of requestors currently using a page

3 Handling Page Requests

When pages are requested from the buffer manager and the page already exists within memory, the page's pin count is incremented and the page's memory address is returned.

If the page does not exist in the buffer pool and there is still space, the next empty frame is found and the page is read into that frame. The page's pin count is set to 1 and the page's memory address is returned. In the case where the page does not exist and there are no empty frames left, a replacement policy must be used to determine which page to evict.

The choice of replacement policy is heavily dependent on page access patterns and the optimal policy is chosen by counting page hits. A **page hit** is when a requested page can be found in memory without having to go to disk. Each **page miss** incurs an additional IO cost, so a good eviction policy is critical for performance. The **hit rate** for an access pattern is defined as # of page hits / (# of page hits + # of page misses) or more simply, # of page hits / # of page accesses.

Additionally, if the evicted page has the dirty bit set, the page is written to disk to ensure that updates are persisted. The dirty bit is set to 1 if and when a page is written to with updates in memory. The dirty bit is set to 0 once the page is written back to disk.

Once the requestor completes its workload, it is responsible for telling the buffer manager to decrement the pin count associated with pages that it previously used.

4 LRU Replacement

A commonly used replacement policy is LRU (Least Recently Used). When new pages need to be read into a full buffer pool, the least recently used unpinned page (pin count = 0) is evicted. To track page usage, a last used column is added to the metadata table and measures the latest time at which a page's pin count is decremented.

Frame ID	Page ID	Dirty Bit	Pin Count	Last Used
0	5	1	3	20
1	3	0	1	32
2	10	1	0	40
3	6	0	0	25
4	1	0	1	15

Implementing LRU normally can be costly. The Clock policy provides an alternative implementation that efficiently approximates LRU using a ref bit (recently referenced) column in the metadata table and a clock hand variable to track the current frame in consideration.

Frame ID	Page ID	Dirty Bit	Pin Count	Ref Bit
0	5	1	3	1
1	3	0	1	1
2	10	1	0	1
3	6	0	0	1
4	1	0	1	0

Clock Hand
2

The Clock policy algorithm treats the metadata table as a circular list of frames. It sets the clock hand to the first unpinned frame upon start and sets the ref bit on each page's corresponding row to 1 when it is initially read into a frame. The policy works as follows when trying to evict:

- Iterate through frames within the table, skipping pinned pages and wrapping around to frame 0 upon reaching the end, until the first unpinned frame with ref bit = 0 is found.
- During each iteration, if the current frame's ref bit = 1, set the ref bit to 0 and move the clock hand to the next frame.
- Upon reaching a frame with ref bit = 0, evict the existing page (and write it to disk if the dirty bit is set; then set the dirty bit to 0), read in the new page, set the frame's ref bit to 1, and move the clock hand to the next frame.

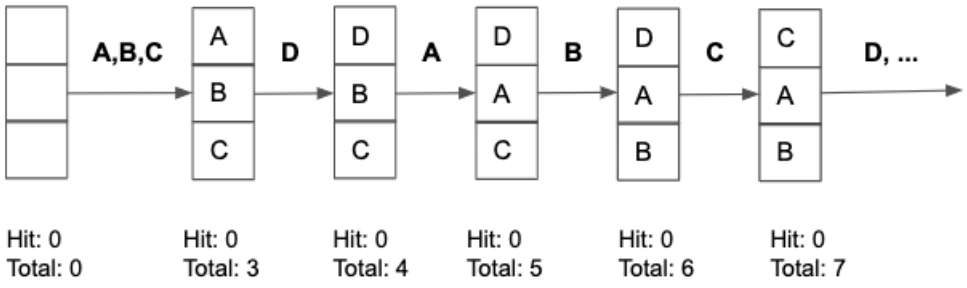
If accessing a page currently in the buffer pool, the clock policy sets the page's ref bit to 1 **without moving the clock hand**.

4.1 Sequential Scanning Performance - LRU

LRU performs well overall but performance suffers when a set of pages S, where $|S| >$ buffer pool size, are accessed multiple times repeatedly.

To highlight this point, consider a 3 frame buffer pool using LRU and having the access pattern:

A B C D A B C D A B C D A B C D



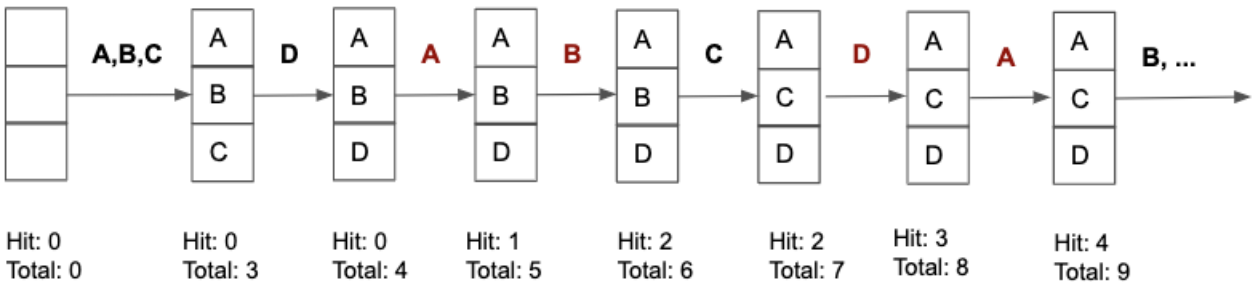
5 MRU Replacement

Another commonly used replacement policy is MRU (Most Recently Used). Instead of evicting the least recently used unpinned page, evict the most recently used unpinned page measured by when the page's pin count was last decremented.

5.1 Sequential Scanning Performance - MRU

At first it might seem counter-intuitive to use this policy but consider the scenario where a 3 frame buffer pool using MRU has the access pattern:

A B C D A B C D A B C D A B C D



Clearly, MRU far outperforms LRU in terms of page hit rate whenever a sequential flooding access pattern occurs.

6 Practice Questions

1. Determine whether each of the following statements is true or false.

   a. Each frame in the buffer pool is the size of a disk page.

   b. The buffer manager code (rather than the file/index management code) is responsible for turning on the dirty bit of a page.

   c. The dirty bit is used to track the popularity of the page.

   d. When using the LRU policy, the reference bits are used to give pages a "second chance" to stay in memory before they are replaced.

   e. A sequential scan of the file will have the same hit rate using either MRU or LRU (starting with an empty buffer pool) when the file is smaller than the buffer pool.

2. For the following question, we are given an initially empty buffer pool with **4** buffer frames. For the access pattern:

   A B D D E A E C A B C A E

   a. What is the hit rate for LRU?

   b. What pages are in the buffer pool when LRU completes?

   c. What is the hit rate for MRU?

   d. What pages are in the buffer pool when MRU completes?

   e. What is the hit rate for Clock?

   f. What pages are in the buffer pool when Clock completes?

7 Solutions

1. a. True. A frame is defined as the size of a disk page.
   b. False. The requester of the page (file/index management code) sets the dirty page.
   c. False. The dirty bit is used to track whether the page has been modified before it's written back to disk. The pin is used to track popularity.
   d. False. Reference bits are not used for LRU policy. They are used for the Clock policy.
   e. True. Since we can fit all pages in memory, we do not need to evict pages during a sequential scan.

2. Detailed access patterns are in diagram below. In LRU and MRU, characters represent misses and asterisks represent hits.
   a. 7/13
   b. A, C, B, E
   c. 7/13
   d. E, B, D, C
   e. 6/13
   f. C, A, B, E

LRU

A					*			*			*	
	B						C			*		
		D	*						B			
				E		*						*

MRU

A					*			*			*	E
	B								*	*		
		D	*									
				E		*	C			*		

Clock (after each access)

```
A: Miss      Clock: (A,1)(_,0)(_,0)(_,0)   Hand: 2
B: Miss      Clock: (A,1)(B,1)(_,0)(_,0)   Hand: 3
D: Miss      Clock: (A,1)(B,1)(D,1)(_,0)   Hand: 4
D: Hit       Clock: (A,1)(B,1)(D,1)(_,0)   Hand: 4
E: Miss      Clock: (A,1)(B,1)(D,1)(E,1)   Hand: 1
A: Hit       Clock: (A,1)(B,1)(D,1)(E,1)   Hand: 1
E: Hit       Clock: (A,1)(B,1)(D,1)(E,1)   Hand: 1
C: Miss      Clock: (C,1)(B,0)(D,0)(E,0)   Hand: 2
A: Miss      Clock: (C,1)(A,1)(D,0)(E,0)   Hand: 3
B: Miss      Clock: (C,1)(A,1)(B,1)(E,0)   Hand: 4
C: Hit       Clock: (C,1)(A,1)(B,1)(E,0)   Hand: 4
A: Hit       Clock: (C,1)(A,1)(B,1)(E,0)   Hand: 4
E: Hit       Clock: (C,1)(A,1)(B,1)(E,1)   Hand: 4
```

CS 186
Spring 2021

Relational Algebra

1 Motivation

In the previous notes we talked about how SQL is a declarative programming language. This means that you specify what you want, but you don't have to specify how to do it. This is great from a user's perspective as it makes the queries much easier to write. As database engineers, however, we often want a language that is more expressive. When we study query optimization in a few weeks we're going to want a way to express the many different valid plans a database can use to execute a query. For this we will use **Relational Algebra**, a procedural programming language (meaning that the query specifies exactly what operators to use and in what order).

2 Relational Algebra Introduction

All of the operators in relational algebra take in a relation and output a relation. A basic query looks like this:

$$\pi_{name}(dogs)$$

The π operator picks only the columns that it wants to advance to the next operator (just like SQL SELECT). In this case, the operator takes the `dogs` relation in as a parameter and returns a relation that only has the `name` column of the `dogs` relation. An important fact about relational algebra is that the relations are sets of tuples, meaning that they cannot have duplicates in them. If the dogs relation is initially:

name	**age**
Scooby	10
Buster	15
Buster	20

The query above would return:

name
Scooby
Buster

Initially the two Busters are different because they have different ages, but once you get rid of the age column, they become duplicates, so only one remains in the output relation.

Let's formally introduce the relational algebra operators.

3 Projection (π)

We have already been introduced to the **projection operator** which selects only the columns specified. The columns are specified in the subscript of the operator like almost all parameters to

operators. The projection operator is relational algebra's version of the SQL SELECT clause.

We now can express SQL queries involving just the SELECT and FROM clauses with relational algebra. For example the SQL query:

```
SELECT name FROM dogs;
```

Can be represented with the expression we introduced in section 2:

$$\pi_{name}(dogs)$$

Note that there is no operator equivalent to the FROM operator in relational algebra because the parameters of these operators specify which tables we pull from.

4 Selection (σ)

The **selection operator** is used to filter rows based on a certain condition. Don't let the name confuse you - this operator is equivalent to SQL's WHERE clause, **not** its SELECT clause. Let's try to express the following query in terms of relational algebra:

```
SELECT name, age FROM dogs WHERE age = 12;
```

The equivalent relational algebra expression is:

$$\sigma_{age=12}(\pi_{name,age}(dogs))$$

Another correct expression for that query is:

$$\pi_{name,age}(\sigma_{age=12}(dogs))$$

This illustrates the beauty of relational algebra. There is only one (reasonable) way to write SQL for what the query is trying to accomplish, but we can come up with multiple different expressions in relational algebra that get the same result. In the first expression we select only the columns we want first, and then we filter out the rows we don't want. In the second we filter the rows first and then select the columns. We will soon learn ways to evaluate which of these plans is better!

The selection operator also supports compound predicates. The $\wedge$ symbol corresponds to the AND keyword in SQL and the $\vee$ symbol corresponds to the OR keyword. For example,

```
SELECT name, age FROM dogs WHERE age = 12 AND name = 'Timmy';
```

is equivalent to

$$\pi_{name,age}(\sigma_{age=12 \wedge name='Timmy'}(dogs))$$

5 Union ($\cup$)

The first way we will learn how to combine data from different relations is with the **union operator**. Just like the UNION clause in SQL, we take all the rows from each tuple and combine them removing duplicates along the way. As an example, say we have a dogs table:

name	**age**
Scooby	10
Buster	15
Garfield	20

and a cats table that looks like this:

name	**age**
Tom	8
Garfield	10

The expression:

$$\pi_{name}(dogs) \cup \pi_{name}(cats)$$

would return:

name
Scooby
Buster
Tom
Garfield

Note that Garfield only shows up once because relations are sets of tuples and remove all duplicates as a result. Additionally, note that one rule for all of these set operators is that they must operate on relations that have the same number of attributes (columns), and the attributes must have the same type. It would not be legal to union a relation with two columns with a relation that only has one column nor would it be legal to union a relation with a column of strings with another relation with one column of ints.

6 Set Difference (-)

Another set operator is the **set difference** operator. Set difference is equivalent to the SQL clause EXCEPT. It returns every row in the first table except the rows that also show up in the second table. If you run:

$$\pi_{name}(dogs) - \pi_{name}(cats)$$

expression on the dogs and cats table introduced in the previous section you would get:

name
Scooby
Buster

Garfield does not show up because he is in the cats table, and none of the cats' names will show up because it is only possible for rows from the first relation to show up in the output.

7 Intersection ($\cap$)

Intersection is just like the INTERSECT SQL operator in that it only keeps rows that occur in both tables in the intersection. If you run:

$$\pi_{name}(dogs) \cap \pi_{name}(cats)$$

on the tables introduced in section 5 you would get:

name
Garfield

because Garfield is the only name to occur in both tables.

8 Cross Product ($\times$)

The cross product operator is just like performing a Cartesian product in SQL. The output is one tuple for every possible pair of tuples, while ensuring that duplicates are removed from the final output as always to satisfy set semantics. As an example, say we have a dogs table:

name	age
Scooby	10
Buster	15
Garfield	20

and a parks table:

park	city
Golden Gate Park	San Francisco
Central Park	New York City

The relational algebra equivalent of

```
SELECT * FROM dogs, parks;
```

is

$$dogs \times parks$$

and the output will be:

name	age	park	city
Scooby	10	Golden Gate Park	San Francisco
Scooby	10	Central Park	New York City
Buster	15	Golden Gate Park	San Francisco
Buster	15	Central Park	New York City
Garfield	20	Golden Gate Park	San Francisco
Garfield	20	Central Park	New York City

In fact, the cross product ($\times$) is the basis for the inner join, which we will go over next.

9 Joins ($\bowtie$)

We haven't yet discussed how to represent joins in relational algebra - let's fix that! To inner join two tables together, write the left table on the left of the $\bowtie$ operator, put the join condition in the subscript, and put the right operator on the right side. To join together the cats table with the dogs table on the name column, you would write:

$$cats \bowtie_{cats.name=dogs.name} dogs$$

If you don't specify the join condition, it becomes a natural join. Recall from the SQL notes that a natural join joins together all columns from each table with the same name. Therefore, you could also write the same query as above like:

$$cats \bowtie dogs$$

Formally, we refer to the inner join operator as a **Theta Join** ($\bowtie_\theta$). The θ refers to the join condition, so for the expression from above, the θ join condition is $cats.name = dogs.name$.

The $\bowtie$ operator performs an inner join, which is the only join we will cover for relational algebra expressions in this class. There are ways to derive right, left, and full outer joins from the operators we have already introduced, but that is beyond the scope of this class.

Just like the selection operator σ, the join operator $\bowtie$ also supports the compound predicate operators $\wedge$ (AND) and $\vee$ (OR).

Theta joins and natural joins can actually be derived from just a cross product ($\times$) and a conjunction of selections (σ). For example,

$$cats \bowtie_\theta dogs$$

can be rewritten as

$$\sigma_\theta(cats \times dogs)$$

and the natural join

$$cats \bowtie dogs$$

can be rewritten as

$$\sigma_{cats.col1=dogs.col1 \wedge ... \wedge cats.colN=dogs.colN}(cats \times dogs)$$

10 Rename (ρ)

The **rename operator** essentially accomplishes the same thing as aliasing in SQL. If you wanted to avoid having to include the table name for the rest of the expression like you would for the expressions in the join section, you could instead write:

$$cats \bowtie_{name=dname} \rho_{name->dname}(dogs)$$

This expression renames the dogs relation's name column to dname first, so there is no conflict in column names. You can no longer use a natural join anymore because the columns do not have the same name, but you no longer need to specify which relation the column is coming from if you want to include other operators.

11 Group By / Aggregation (γ)

The final relational algebra operator we will cover is the **groupby / aggregation operator**, which is essentially equivalent to using the GROUP BY and HAVING clauses in SQL. For example, the SQL query

```
SELECT * FROM dogs GROUP BY age HAVING COUNT(*) > 5;
```

can be expressed in relational algebra as

$$\gamma_{age,COUNT(*)>5}(dogs)$$

Furthermore, the γ operator can be used to select aggregate columns, such as MAX, MIN, SUM, COUNT, etc. from SQL. This modified query from earlier

```
SELECT age, SUM(weight) FROM dogs GROUP BY age HAVING COUNT(*) > 5;
```

can be expressed in relational algebra as

$$\gamma_{age,SUM(weight),COUNT(*)>5}(dogs)$$

Relational Algebra

12 Practice Questions

Given the following two relations:

```
teams(teamid, name)
players(playerid, name, teamid, position)
```

Answer the following questions:

1. Write an expression that finds the name and playerid of every player that plays the "center" position.
2. Write an expression that finds the name of every player that plays on the "Warriors". How would this expression change if we renamed players' teamid column to pteamid?
3. Write an expression that finds the teamid of all teams that do not have any players.
4. Write an expression that is equivalent to the following SQL query:

```
SELECT teamid AS tid
FROM players
WHERE players.teamid NOT IN
    (SELECT teamid FROM teams)
AND position='shooting guard';
```

13 Solutions

1. $\pi_{name,playerid}(\sigma_{position='center'}(players))$. We first filter out the rows for players who aren't centers, then we project only the columns that we need.
2. $\pi_{players.name}(\sigma_{teams.name='Warriors'}(teams \bowtie_{teams.teamid=players.teamid} players))$. We first join together the teams and players table to get all the information that we need, then we filter out the rows that aren't for players who play for the Warriors, then we finally project the only column that we're looking for.
3. $\pi_{teamid}(teams) - \pi_{teamid}(players)$. All teams must be in the teams table so we first get all their teamids. Then we subtract any teamid that appears in the players table, because if that teamid appears in the players table it implies that the team has a player on it. We are then left with only teamids of teams that don't have any players.
4. $\rho_{teamid \to tid}(\pi_{teamid}(\sigma_{position='shooting\ guard'}(players)) - \pi_{players.teamid}(players \bowtie_{players.teamid=teams.teamid} teams))$
   We first filter out rows for players who aren't shooting guards, then we only project the column we need, teamid. We then use set difference to only keep players who play for a team not in the teams table. Finally, we use the renaming operator to rename teamid to tid.

CS 186
Spring 2021

Sorting

In CS61B, you learned about many different sorting algorithms. Why are we learning yet another new one in this class? All of the traditional sorting algorithms (i.e. quick sort, insertion sort, etc.) rely on us being able to store all of the data in memory. This is a luxury we do not have when developing a database. In fact, most of the time our data will be an order of magnitude larger than the memory available to us.

1 I/O Review

Remember that we incur 1 I/O any time we either write a page from memory to disk or read a page from disk into memory. Because of how time consuming it is to go to disk, we only look at the number of I/Os an algorithm incurs when analyzing its performance. We can pretty much ignore traditional measures of algorithmic complexity like big-O. Therefore, when developing our sorting algorithm we will attempt to minimize the number of I/Os it will incur. One last thing to note when counting I/Os is that we ignore any potential caching done by the buffer manager. This implies that once we unpin the page and say that we are done using it, the next time we attempt to access the page it will always cost 1 I/O.

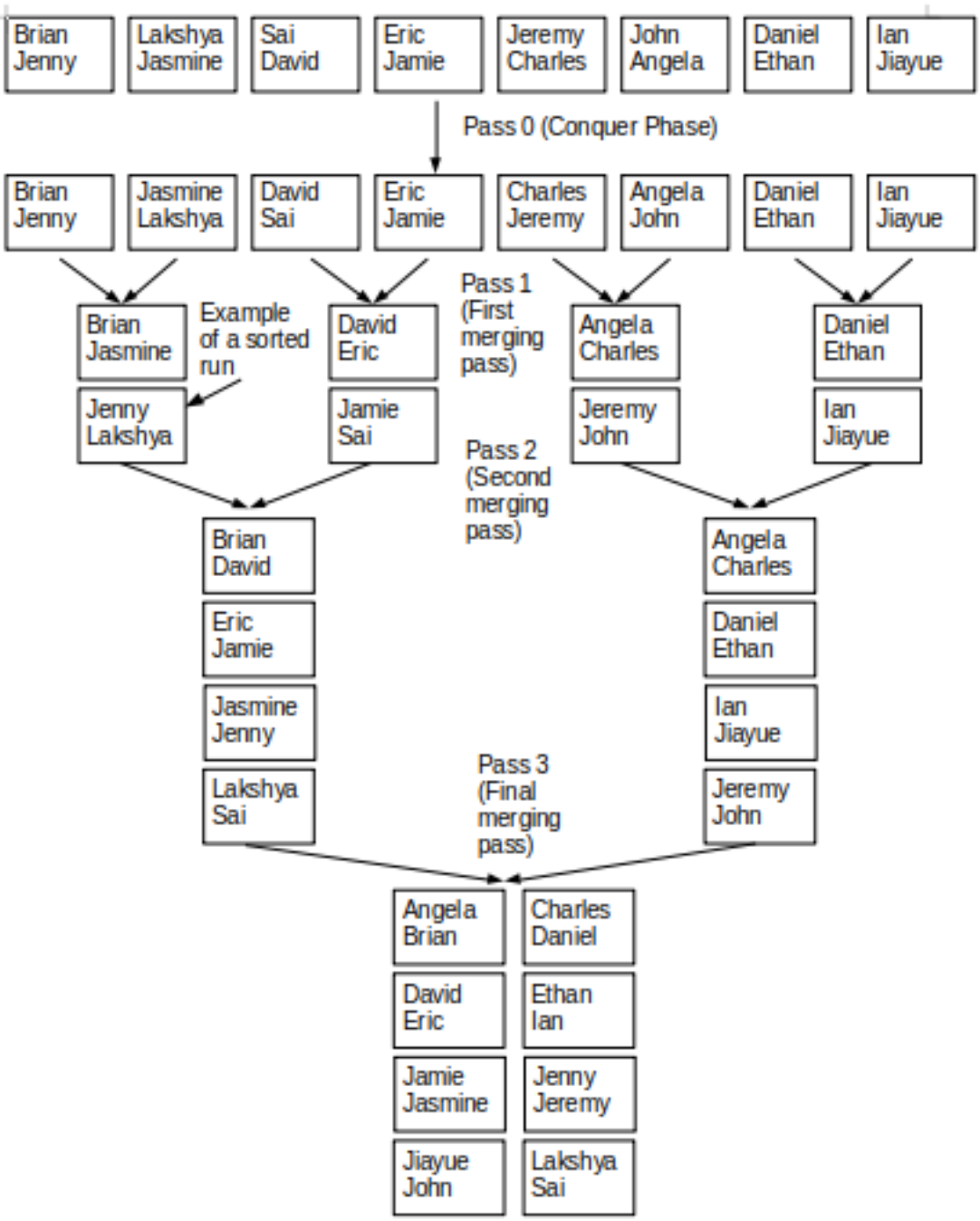
2 Two Way External Merge Sort

Let's start by developing a sorting algorithm that works but is not as good as possible. Because we cannot keep all of our data in memory at one time, we know that we are going to sort different pieces of it separately and then merge it together.

In order to merge two lists together efficiently, they must be sorted first. This is a hint that the first step of our sorting algorithm should be to sort the records on each individual page. We'll call this first phase the "conquer" phase because we are conquering individual pages.

After this, let's start merging the pages together using the merge algorithm from merge sort. We'll call the result of these merges **sorted runs**. A sorted run is any sequence of pages that is sorted.

The rest of the algorithm will simply be to continue merging these sorted runs until we have only one sorted run remaining. One sorted run implies that our data is fully sorted! See the image on the next page for a diagram of the algorithm run to completion.


Brian Jenny
Lakshya Jasmine
Sai David
Eric Jamie
Jeremy Charles
John Angela
Daniel Ethan
Ian Jiayue
Pass 0 (Conquer Phase)
Brian Jenny
Jasmine Lakshya
David Sai
Eric Jamie
Charles Jeremy
Angela John
Daniel Ethan
Ian Jiayue
Pass 1 (First merging pass)
Example of a sorted run
Brian Jasmine
Jenny Lakshya
David Eric
Jamie Sai
Angela Charles
Jeremy John
Daniel Ethan
Ian Jiayue
Pass 2 (Second merging pass)
Brian David
Eric Jamie
Jasmine Jenny
Lakshya Sai
Angela Charles
Daniel Ethan
Ian Jiayue
Jeremy John
Pass 3 (Final merging pass)
Angela Brian
David Eric
Jamie Jasmine
Jiayue John
Charles Daniel
Ethan Ian
Jenny Jeremy
Lakshya Sai

3 Analysis of Two Way Merge

When analyzing a database algorithm, the most important metric is the number of I/Os the algorithm takes, so let's start there. First, notice that each pass over the data will take $2 * N$ I/Os where N is the number of data pages. This is because for each pass, we need to read in every page and write back every page after modifying it.

The only thing left to do is to figure out how many passes we need to sort the table. We always need to do that initial "conquer" pass, so we always have at least one. Now, how many merging passes are required? Each pass, we cut the amount of sorted runs we have left in half. Dividing the data each time should scream out *log* to you, and because we're diving it by 2, the base of our log will be 2. Therefore we need $\lceil \log_2(N) \rceil$ merging passes, and $1 + \lceil \log_2(N) \rceil$ passes in total. This leads to our final formula of $2N * (1 + \lceil \log_2(N) \rceil)$ I/Os.

Now let's analyze how many buffer pages we need to execute this algorithm. Remember that a **buffer page**, or **buffer frame**, is a slot for a page to be stored in memory.

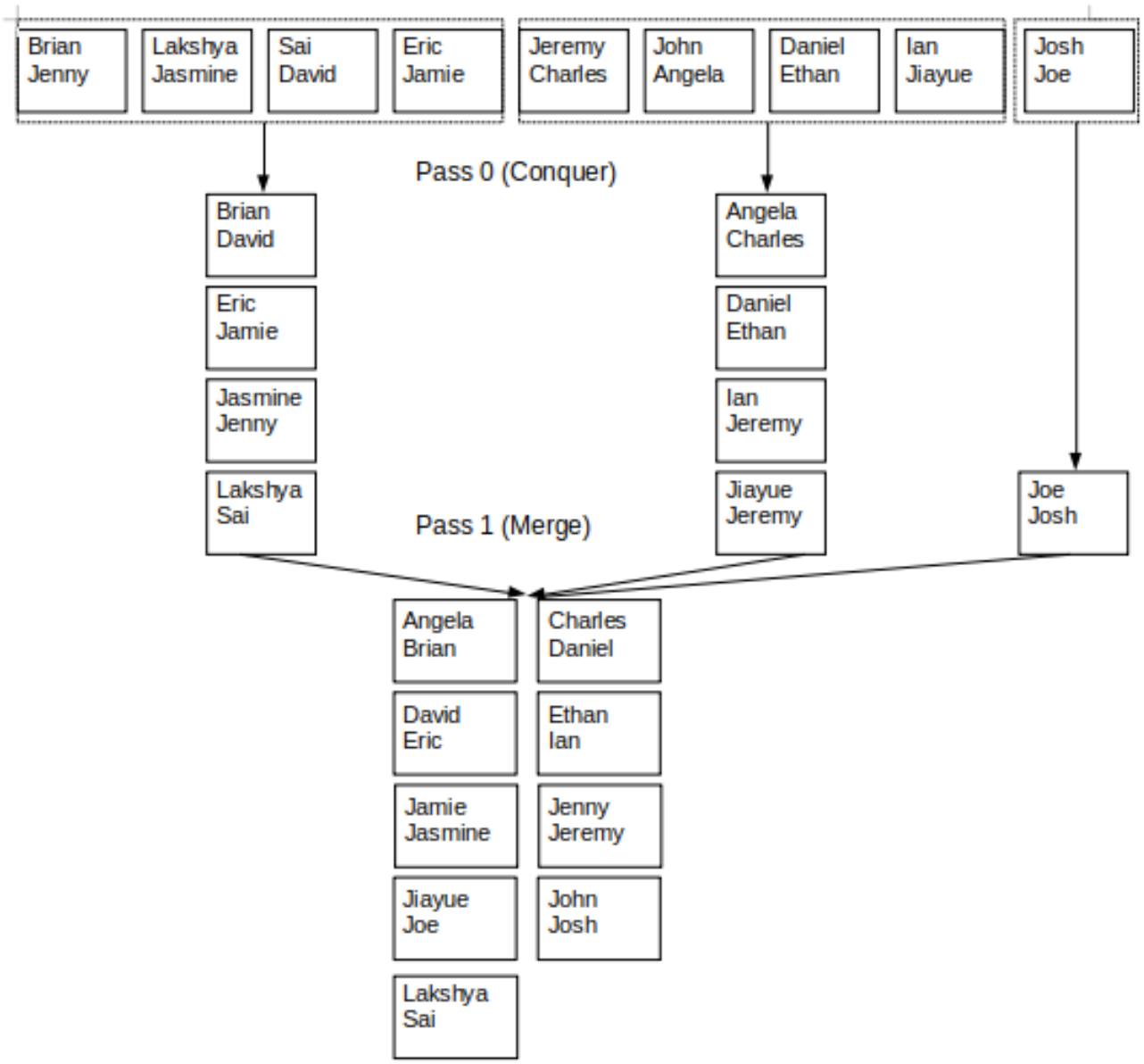
The first pass, the "conquering pass", sorts each page individually. This means we only ever need 1 buffer page for this pass, to hold the page that we are sorting!

Now let's analyze the merging passes. Recall how merging works in merge sort. We only compare the first value for the two lists that we are merging. This means that we only need to store the first page of each sorted run in memory, rather than the entire sorted runs. When we have used all of the records from the original page, we simply get rid of that page from memory and load in the next page of the sorted run. So far, we need 2 buffer pages (1 for each sorted run). We will call the buffer frame used to store the front of each sorted run the **input buffer**. The only thing we're missing now is a place to store our output. We need to write out records somewhere, so we need 1 more page, called the **output buffer**. Once this page has filled up, we flush it to disk and start constructing the next page. In total, we have two input buffers and 1 output buffer for a total of 3 pages required. This does not take advantage of all the memory that we have. Let's construct a better algorithm that uses all of our memory.

4 Full External Sort

Let's assume we have B buffer pages available to us. The first optimization we will make is in the initial "conquer pass." Rather than just sorting individual pages, let's load B pages and sort them all at once. This way we will produce fewer and longer sorted runs after the first pass.

The second optimization is to merge more than 2 sorted runs together at a time. We have B buffer frames available to us, but we need 1 for the output buffer. This means that we can have B-1 input buffers and can thus merge together B-1 sorted runs at a time. See the next page for a diagram of this sort assuming we have 4 buffer frames available to us.



Now we take in 4 pages at a time during the conquering phase and output a sorted run of length 4. In the merging pass, we can merge all three sorted runs produced during the conquering pass at once. This cut the number of passes (and thus our I/Os) in half!

5 Analysis of Full External Merge Sort

Let's now figure out how many I/Os our improved sort takes using the same process we did for Two-Way Merge. The conquering pass produces only $\lceil N/B \rceil$ sorted runs now, so we have fewer runs to merge. During merging, we are dividing the number of sorted runs by $B-1$ instead of 2, so the base of our log needs to change to $B-1$. This makes our overall sort take $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$ passes, and thus $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$ I/Os.

6 Practice Questions

1) You are trying to sort the Students table which has 1960 pages with 8 available buffer pages.
a. How many sorted runs will be produced after each pass?
b. How many pages will be in each sorted run for each pass?
c. How many I/Os does the entire sorting operation take?

2) What is the minimum number of buffer pages that we need to sort 1000 data pages in two passes?

3) You are trying to sort the Sailors table which has 200 pages. Suppose that during Pass 0, you have 10 buffer pages available to you, but for Pass 1 and onwards, you only have 5 buffer pages available.
a. How many sorted runs will be produced after each pass?
b. How many pages will be in each sorted run for each pass?
c. How many I/Os does the entire sorting operation take?

7 Solutions

1) The first pass loads all 8 buffer pages with data pages at a time and outputs sorted runs until each page is part of a sorted run. This means that we will have 1960 / 8 = 245 sorted runs of 8 pages after pass 0.

All subsequent passes merge 7 sorted runs at a time (remember we need a frame for the output buffer) so after the first sorting pass we will have 245 / 7 = 35 sorted runs of 8 * 7 = 56 pages.

The next merging pass produces 35 / 7 = 5 sorted runs of 56 * 7 = 392 pages.

The next merging pass can merge all remaining sorted runs (because there are ≤ 7 sorted runs) so it will produce 1 sorted run of all 1960 pages.

Therefore the answer to a is: **245, 35, 5, 1** and the answer to b is: **8, 56, 392, 1960**.

Each pass takes 2 * N I/Os where N is the total number of data pages because each page gets read and written in a pass. This means that the answer to c is: 4 * 2 * 1960 = **15,680 I/Os**.

2) The conquer pass of sorting divides the number of sorted runs that we have by B (initially we consider each page to be its own sorted run even if it isn't actually sorted yet). The merging pass divides the number of sorted runs by B-1. Together, two passes will divide the number of sorted runs we have by B(B-1). After those two passes we need to have only one sorted run remaining, so we need:

$$\frac{1000}{B(B-1)} \leq 1$$

You can move the denominator to the other side and then move 1000 over as well to get:

$$B^2 - B - 1000 \geq 0$$

Using the quadratic formula, you can get that B = 32.1 which means we need **33 buffer pages**.

3) In Pass 0, we load all 10 buffer pages at a time with data pages and produce sorted runs. Since we have 200 data pages, we will have 200/10 = 20 sorted runs after Pass 0 of 10 pages each.

In Pass 1, we can now only use 5 buffer pages. Here, we will merge together sorted runs together, reserving 1 output buffer and designating the remaining 4 buffer pages as input buffers. Thus, we can merge together 4 sorted runs at a time. Since we start with 20 sorted runs, we will end up with 20/4 = 5 sorted runs of 10 * 4 = 40 pages after Pass 1.

In Pass 2, we can merge up to 4 sorted runs at a time, so we will end up with 2 sorted runs (1 with 160 pages, 1 with 40 pages).

In Pass 3, we merge all remaining sorted runs, producing 1 sorted run of 200 pages.
Therefore the answer to a is: **20, 5, 2, 1** and the answer to b is: **10, 40, (160 and 40), 200**.

Each pass takes 2 * N I/Os where N is the total number of data pages because each page gets read and written in a pass. This means that the answer to c is: 4 * 2 * 200 = **1600 I/Os**.

CS 186
Spring 2021

Hashing

1 Motivation

Sometimes, sorting is a bit overkill for the problem. In a lot of cases, all we want is to group the same value together, but we do not actually care about the order the values appear in (think GROUP BY or de-duplication). In a database, grouping like values together is called hashing. We cannot build a hash table in the standard way you learned in 61B for the same reason we could not use quick sort in the last note; we cannot fit all of our data in memory! Let's see how to build an efficient out-of-core hashing algorithm.

2 General Strategy

Because we cannot fit all of the data in memory at once, we'll need to build several different hash tables and concatenate them together. There is a problem with this idea though. What happens if we build two separate hash tables that each have the same value in them (e.g. "Brian" occurs in both tables)? Concatenating the the tables will result in some of the "Brian"s not being right next to each other.

To fix this, before building a hash table out of the data in memory, we need to guarantee that if a certain value is in memory, all of its occurrences are also in memory. In other words, if "Brian" occurs in memory at least once, then we can only build the hash table if every occurrence of "Brian" in our data is currently in memory. This ensures that values can only appear in one hash table, making the hash tables safe to concatenate.

3 The Algorithm

We will use a divide and conquer algorithm to solve this problem. The "divide" phase will be partitioning passes, and the "conquer" phase will be actually constructing the hash tables. Just like in the sorting note, we will assume that we have B buffer frames available to us.

The first partitioning pass will hash each record to $B-1$ **partitions**. A partition is a set of pages such that the values on the pages all hash to the same value (for the hash function used to construct the partition). We do this by using $B-1$ output buffers. When an output buffer fills up we flush the page to disk. When that buffer fills up the next time, we place it adjacent to the page we flushed to disk before from that same buffer. The most important property of each partition is that if a certain value appears in that partition, all occurrences of that value in our data appear in that partition. In other words, if "Brian" appears in that partition, "Brian" will not appear in any other partition. This is because "Brian" always hashes to the same value, so it cannot possibly end up in a different partition. We only have $B-1$ partitions because we need to save one buffer frame to be the input buffer.

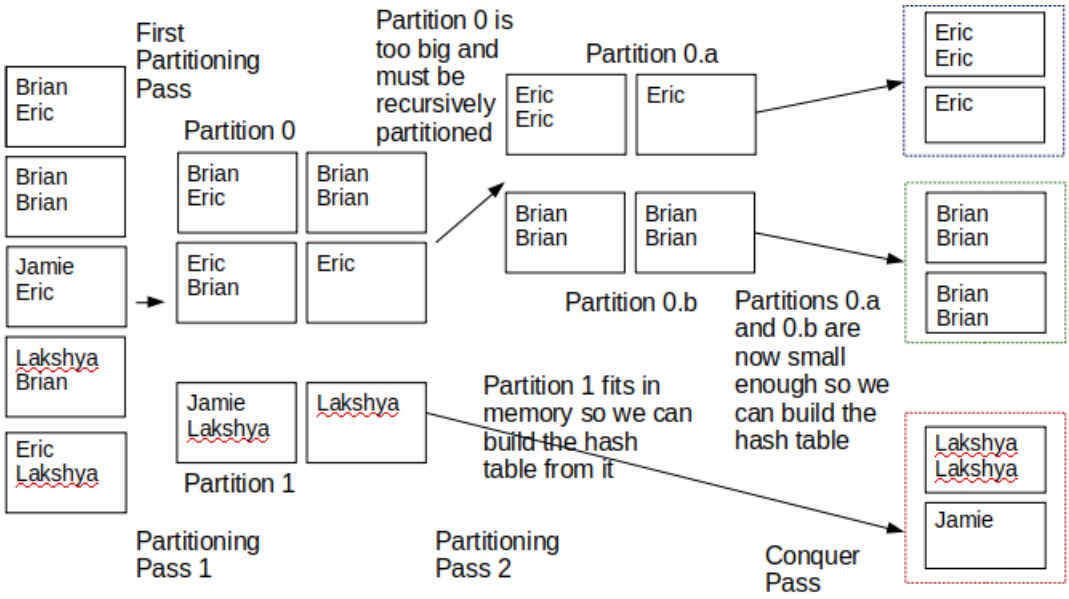
After this first partitioning pass, we can go right to the hash table building phase for the partitions that fit in memory. What does it mean to fit in memory? Fitting in memory means the partition

must be B pages big or less. For the partitions that are too big, we simply repartition them using a different hash function than we used in the first pass. Why a different hash function? If we reused the original function, every value would hash to its original partition so the partitions would not get any smaller. We can recursively partition as many times as necessary until all of the partitions have at most B pages.

Now all of our partitions can fit in memory, and we know that all like values occur in the same partition. The only thing left to do is build a hash table for each partition and write each hash table to disk.

4 Example

In the following example, we will assume that we have $B = 3$ buffer pages available to us. We also assume that Brian and Eric hash to the same value for the first hash function but different values for the second hash function, and Jamie and Lakshya hash to the same value for the first hash function.



You can see that Partition 0 is too big because it contains 4 pages, but we only have 3 buffer frames available to us. When it gets recursively partitioned, however, the subpartitions (0.a and 0.b) are both only 2 pages long, so they can now fit into memory. You can also see that after the final "conquer" pass, all of the like values are next to each other, which is our end goal.

5 Analysis of External Hashing

We're not going to be able to create a simple formula to count the number of I/Os like in the sorting algorithm because we do not know how large the partitions will be. One of the first things we need to recognize is that it is possible for the number of pages in the table to increase after a partitioning pass. To see why, consider the following table in which we can fit two integers on a page:

[1, 2] [1, 4] [3, 4]

Let's assume B=3, so we only divide the data into 2 partitions. Let's assume 1 and 3 hash to partition 1, and 2 and 4 hash to partition 2. After partitioning, partition 1 will have:

[1, 1], [3]

And partition 2 will have:

[4, 2], [4]

Notice that we now have 4 pages when we only started with 3. Therefore, the only reliable way to count the number of I/Os is to go through each pass and see exactly what will be read and what will be written. Let m be the total number of partitioning passes required, let r_i be the number of pages you need to read in for partitioning pass i, let w_i be the number of pages you need to write out of partitioning pass i, and let X be the total number of pages we have after partitioning that we need to build our hash tables out of. Here is a formula for the number of I/Os:

$$(\sum_{i=1}^{m} r_i + w_i) + 2X$$

The summation doesn't tell us anything that we didn't already know; we need to go through each pass and figure out exactly what was read and written. The final $2X$ part, says that in order to build our hash tables, we need to read and write every page that we have after the partitioning passes.

Here are some important properties:

1. $r_0 = N$
2. $r_i \leq w_i$
3. $w_i \geq r_{i+1}$
4. $X \geq N$

Property 1 says that we must read in every page during the first partitioning pass. This comes straight from the algorithm.

Property 2 says that during a partitioning pass we will write out at least as many pages as we read in. This comes directly from the explanation above - we may create additional pages during a partitioning pass.

Property 3 says that we will not read in more pages than what we wrote out during the partitioning pass before. In the worst case, every partition from pass i will need to be repartitioned, so this would require us to read in every page. In most cases, however, some partitions will be small enough to fit in memory, so we can read in fewer pages than we produced during the previous pass.

Property 4 says that the number of pages we will build our hash table out of is at least as big as the number of data pages we started with. This comes from the fact that the partitioning passes can only increase the number of data pages, not decrease them.

6 Practice Questions

1) How many IOs does it take to hash a 500 page table with B = 10 buffer pages? Assume that we use perfect hash functions.

2) We want to hash a 30 page table using B=6 buffer pages. Assume that during the first partitioning pass, 1 partition will get 10 data pages and the rest of the pages will be divided evenly among the other partitions. Also assume that the hash function(s) we use for recursive partitioning are perfect. How many IOs does it take to hash this table?

3) If we had 20 buffer pages to externally hash elements, what is the minimum number of pages we could externally hash to guarantee that we would have to use recursive partitioning?

7 Solutions

1) The first partitioning pass divides the 500 pages into 9 partitions. This means that each partition will have 500 / 9 = 55.6 $\implies$ 56 pages of data. We had to read in the 500 original pages, but we have to write out a total of 56 * 9 = 504 because each partition has 56 pages and there are 9 partitions. The total number of IOs for this pass is therefore 500 + 504 = 1004.

We cannot fit any partition into memory because they all have 56 pages, so we need to recursively partition all of them. On the next partitioning pass each partition will be divided into 9 new partitions (so 9*9 = 81 total partitions) with 56 / 9 = 6.22 $\implies$ 7 pages each. This pass needed to read in the 504 pages from the previous pass and write out 81 * 7 = 567 pages for a total of 1071 IOs.

Now each partition is small enough to fit into memory. The final conquer pass will read in each partition from the previous pass and write it back out to build the hash table. This means every page from the previous pass is read and written once for a total of 567 + 567 = 1134 IOs.

Adding up the IOs from each pass gives a total of 1004 + 1071 + 1134 = **3209 IOs**

2) There will be 5 partitions in total (B-1). The first partition gets 10 pages, so the other 20 will be divided over the other 4 partitions, meaning each of those partitions gets 5 buffer pages. We had to read in all 30 pages to partition them, and we write out 5 * 4 + 10 = 30 pages as well, meaning the first partitioning pass takes 60 IOs.

The partitions that are 5 pages do not need to be recursively partitioned because they fit in memory. The partition that is 10 pages long, however, does. This partition will be repartitioned into 5 new partitions of size 10/5 = 2 pages. We read in all 10 pages for this partition, and we wrote out 5 * 2 = 10 pages, meaning that this recursive partitioning pass takes 20 IOs.

The final conquer pass needs to read in and write out every partition, so it will take 4 * 5 * 2 = 40 IOs to conquer the partitions that did not need to be repartitioned and 5 * 2 * 2 = 20 IOs to conquer the partitions that were created from recursive partitioning, for a total of 60 IOs.

Finally, add up the IOs from each pass and get a total of 60 + 20 + 60 = **140 IOs**.

3) **381 pages**. Since B = 20, we can potentially hash upto B * (B - 1) = 20 * (20 - 1) = 380 pages without doing recursive partitioning. If we want to guarantee recursive partitioning, we need one more page then this, giving us 381 pages as our solution.

Even with a perfect hash function, one page would have B+1 pages, which would require recursive partitioning since B+1 can't fit into memory when there are B buffer pages.

CS 186
Spring 2021

Iterators and Joins

1 Introduction

Let's begin with the simplest question: what, exactly, is a join? If you remember the SQL project, you'll remember writing things like `R INNER JOIN S ON R.name = S.name` and other similar statements.

What that actually meant is that you take two relations, `R` and `S`, and create one new relation out of their matches on the join condition – that is, for each record r_i in `R`, find all records s_j in `S` that match the join condition we have specified and write $< r_i, s_j >$ as a new row in the output (all the fields of r followed by all the fields of s). The SQL lecture slides are a great resource for more clarifications on what joins actually are.[1]

Before we get into the different join algorithms, we need to discuss what happens when the new joined relation consisting of $< r_i, s_j >$ is formed. Whenever we compute the cost of a join, we will ignore the cost of writing the joined relation to disk. This is because we are assuming that the output of the join will be consumed by another operator involved later on in the execution of the SQL query. Often times this operator can directly consume the joined records from memory. Don't worry if this sounds confusing right now; we will revisit it in the Query Optimization module, but the important thing to remember for now is that the final write cost is not included in our join cost models!

[1] **notation aside**: `[T]` is the number of pages in table `T`, ρ_T is the number of records on each page of `T`, and `|T|` is the total number of records in table `T`. In other words, `|T|` = `[T]` $\times$ ρ_T. This is really essential to understand the following explanations.

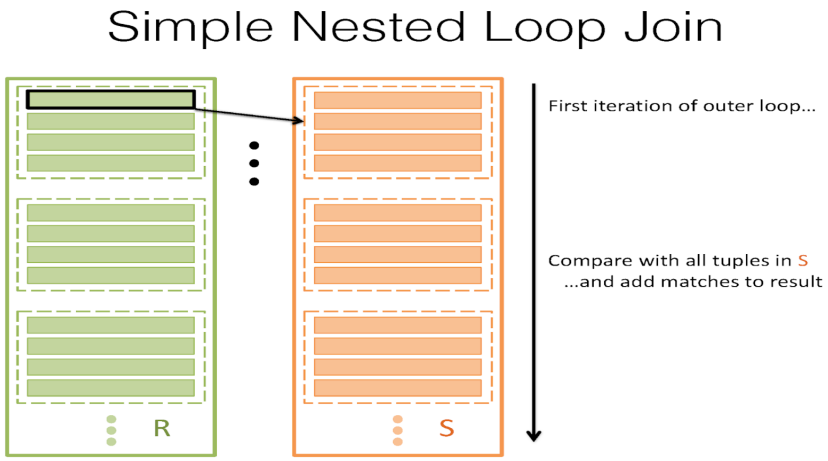
Iterators and Joins

2 Simple Nested Loop Join

Let's start with the simplest strategy possible. Let's say we have a buffer of `B` pages, and we wish to join two tables, `R` and `S`, on the join condition θ. Starting with the most naïve strategy, we can take each record in `R`, search for all its matches in `S`, and then we yield each match.

This is called **simple nested loop join (SNLJ)**. You can think of it as two nested for loops:

```
for each record r_i in R:
    for each record s_j in S:
        if θ(r_i,s_j):
            yield <r_i, s_j>
```



This would be a great thing to do, but the theme of the class is really centered around optimization and minimizing I/Os. For that, this is a pretty poor scheme, because we take each record in `R` and read in every single page in `S` searching for a match. The I/O cost of this would then be $[R]+|R|[S]$, where `[R]` is the number of pages in `R` and `|R|` is the number of records in `R`. And while we might be able to optimize things a slight amount by switching the order of `R` and `S` in the for loop, this really isn't a very good strategy.

Note: SNLJ does *not* incur `|R|` I/Os to read every record in R. It will cost `[R]` I/Os because it's really doing something more like "for each page p_r in `R`: for each record r in p_r: for each page p_s in `S`: for each record s in p_s: join" since we can't read less than a page at a time.

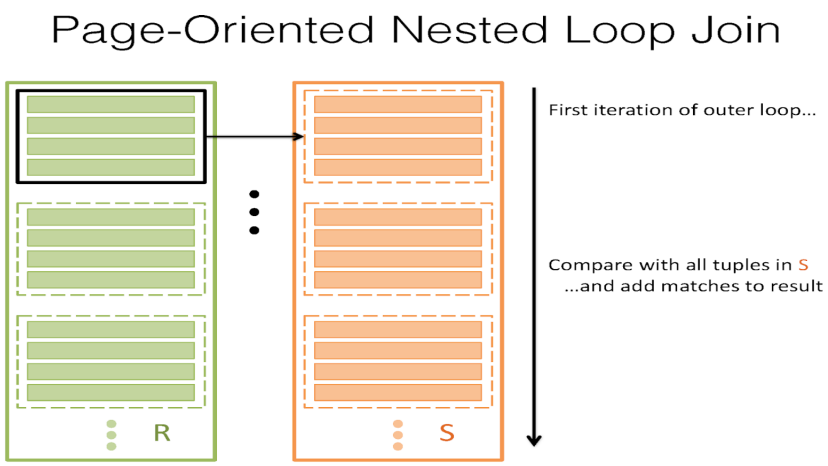
Iterators and Joins

3 Page Nested Loop Join

It's clear that we don't want to read in every single page of `S` for each record of `R`, so what can we do better? What if we read in every single page in `S` for every single page of `R` instead? That is, for a page of `R`, take all the records and match them against each record in `S`, and do this for every page of `R`.

That's called **page nested loop join (PNLJ)**. Here's the pseudocode for it:

```
for each page p_r in R:
    for each page p_s in S:
        for each record r_i in p_r:
            for each record s_j in p_s:
                if θ(r_i, s_j):
                    yield <r_i, s_j>
```



The I/O cost of this is somewhat better. It's $[R] + [R][S]$ – this can be optimized by keeping the smaller relation between `R` and `S` as the outer one.

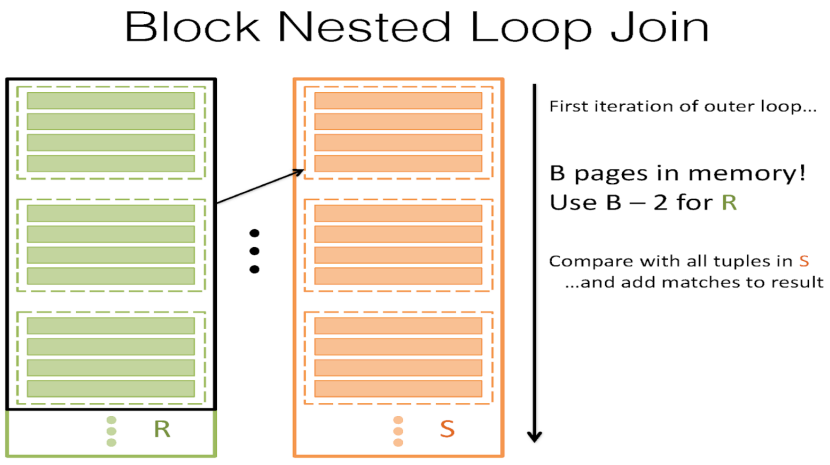
Iterators and Joins

4 Block Nested Loop Join

Page Nested Loop Join is a lot better! The only problem is that we're still not fully utilizing our buffer as powerfully as we can. We have `B` buffer pages, but our algorithm only uses 3 – one for `R`, one for `S`, and one for the output buffer. Remember that the fewer times we read in `S`, the better – so if we can reserve `B-2` pages for `R` instead and match `S` against every record in each "chunk", we could cut down our I/O cost drastically!

This is called **Chunk Nested Loop Join (or Block Nested Loop Join)**. The key idea here is that we want to utilize our buffer to help us reduce the I/O cost, and so we can reserve as many pages as possible for a chunk of `R` – because we only read in each page of `S` once per chunk, larger chunks imply fewer I/Os. For each chunk of `R`, match all the records in `S` against all the records in the chunk.

```
for each block of B−2 pages B_r in R:
    for each page p_s in S:
        for each record r_i in B_r:
            for each record s_j in p_s:
                if θ(r_i,s_j):
                    yield <r_i, s_j>
```



Then, the I/O cost of this can be written as $[R] + \lceil \frac{[R]}{B-2} \rceil [S]$.

This is a lot better! Now, we're taking advantage of our `B` buffer pages to reduce the number of times we have to read in `S`.

5 Index Nested Loop Join

There are times, however, when Block Nested Loop Join isn't the best thing to do. Sometimes, if we have an index on `S` that is on the appropriate field (i.e. the field we are joining on), it can be very fast to look up matches of r_i in `S`. This is called index nested loop join, and the pseudocode goes like this:

```
for each record r_i in R:
    for each record s_j in S where θ(r_i,s_j)==true:
        yield <r_i, s_j>
```

The I/O cost is $[R] + |R| *$(cost to look up matching records in `S`).

The cost to look up matching records in `S` will differ based on the type of index. If it is a B+ tree, we will search starting at the root and count how many I/Os it will take to get to a corresponding record. See the *Clustering* and *Counting I/O's* sections of the B+ tree course notes.

6 Hash Join

Notice that in this entire sequence, we're really trying to look for matching records. Hash tables are really nice for looking up matches, though; even if we don't have an index, we can construct a hash table that is `B-2` pages[2] big on the records of `R`, fit it into memory, and then read in each record of `S` and look it up in `R`'s hash table to see if we can find any matches on it. This is called **Naive Hash Join**. Its cost is $[R] + [S]$ I/Os.

That's actually the best one we've done yet. It's efficient, cheap, and simple. There's a problem with this, however; this relies on `R` being able to fit entirely into memory (specifically, having `R` being $\leq B - 2$ pages big). And that's often just not going to be possible.

To fix this, we repeatedly hash `R` and `S` into `B-1` buffers so that we can get partitions that are $\leq B - 2$ pages big, enabling us to fit them into memory and perform a Naive Hash Join. More specifically, consider each pair of corresponding partitions $\texttt{R}_i$ and $\texttt{S}_i$ (i.e. partition i of `R` and partition i of `S`). If $\texttt{R}_i$ and $\texttt{S}_i$ are both $>$ `B-2` pages big, hash both partitions into smaller ones. Else, if either $\texttt{R}_i$ or $\texttt{S}_i \leq$ `B-2` pages, stop partitioning and load the *smaller* partition into memory to build an in-memory hash table and perform a Naive Hash Join with the larger partition in the pair.

This procedure is called **Grace Hash Join**, and the I/O cost of this is: the cost of hashing plus the cost of Naive Hash Join on the subsections. The cost of hashing can change based on how many times we need to repeatedly hash on how many partitions. The cost of hashing a partition `P` includes the I/O's we need to read all the pages in `P` and the I/O's we need to write all the resulting partitions after hashing partition `P`.

The Naive Hash Join portion cost per partition pair is the cost of reading in each page in both partitions after you have finished.

Grace Hash is great, but it's really sensitive to key skew, so you want to be careful when using this algorithm. Key skew is when we try to hash but many of the keys go into the same bucket. Key skew happens when many of the records have the same key. For example, if we're hashing on the column which only has "yes" as values, then we can keep hashing but they will all end up in the same bucket no matter which hash function we use.

[2] We need one page for the current page in `S` and one page to store output records. The other `B-2` pages can be used for the hash table.

Iterators and Joins

7 Sort-Merge Join

There's also times when it helps for us to sort R and S first, especially if we want our joined table to be sorted on some specific column. In those cases, what we do is first sort `R` and `S`. Then:

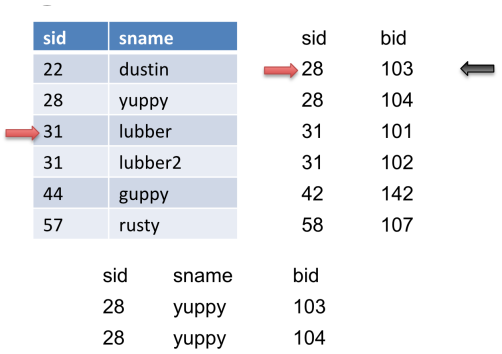
(1) we begin at the start of `R` and `S` and advance one or the other until we get to a match (if $r_i < s_j$, advance `R`; else if $r_i > s_j$, advance `S` – the idea is to advance the lesser of the two until we get to a match).

(2) Now, let's assume we've gotten to a match. Let's say this pair is r_i, s_j. We mark this spot in `S` as $marked(S)$ and check each subsequent record in `S` (s_j,s_{j+1},s_{j+2}, etc) until we find something that is not a match (i.e. read in all records in `S` that match to r_i).

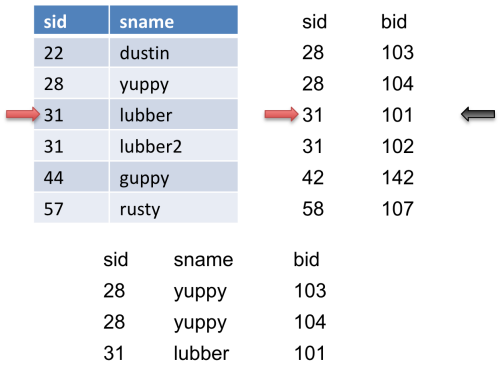
(3) Now, go to the next record in `R` and go back to the marked spot in `S` and begin again at step 1 (except instead of beginning at the start of `R` and the start of `S`, do it at the indices we just indicated) – the idea is that because `R` and `S` are sorted, any match for any future record of `R` cannot be before the marked spot in `S`, because this record $r_{i+1} \geq r_i$ – if a match for r_i did not exist before $marked(S)$, a match for r_{i+1} cannot possibly be before $marked(S)$ either! So we scroll from the marked spot in `S` until we find a match for r_{i+1}.

This is called **Sort-Merge Join** and the average I/O cost is: cost to sort `R` + cost to sort `S` + $([R] + [S])$ (though it is important to note that this is not the worst case!). In the worst case, if each record of `R` matches every record of `S`, the last term becomes $|R| * [S]$. The worst case cost is then: cost to sort `R` + cost to sort `S` + $([R] + |R| * [S])$. That generally doesn't happen, though).

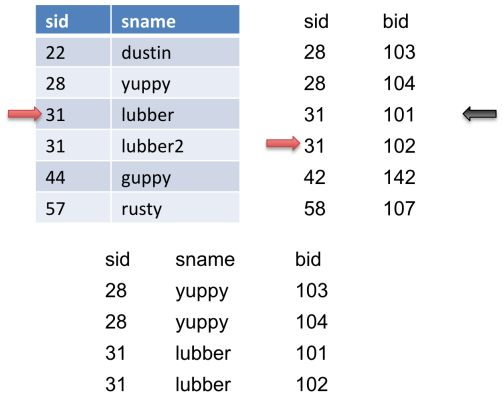
Let's take a look at an example. Let the table on the left be R and the table on the right be S.



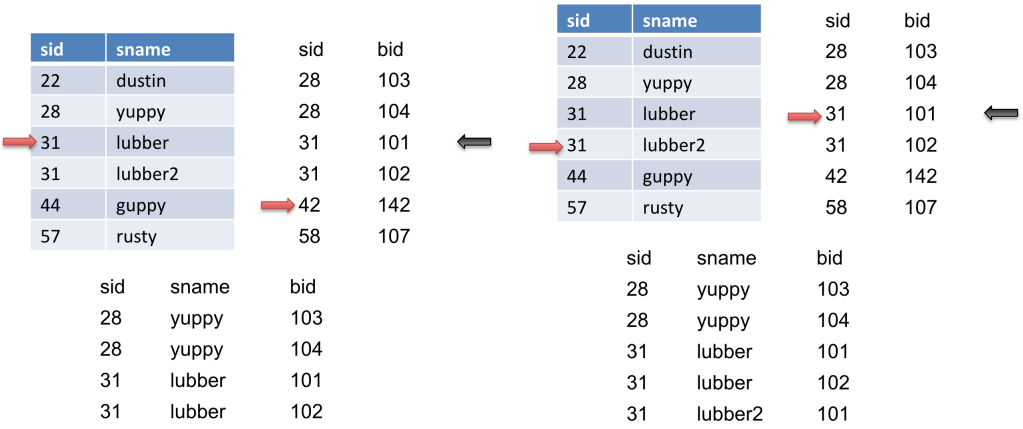
We will advance the pointer (the red arrow) on S because $28 < 31$ until S gets to sid of 31. Then we will mark this record (the black arrow). In addition, we will output this match.



Then we will advance the pointer on S again and we get another match and output it.



We advance the pointer on S again, but we do not get a match. We then reset S to where we marked (the black arrow) and then advance R. When we advance R, we get another match so we output it.



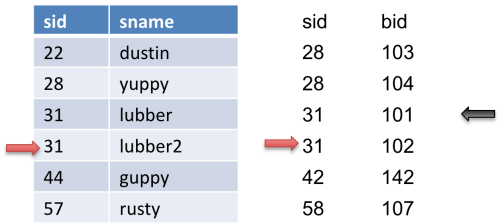
sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101

We then advance S, we get another match so we output it.



sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

```
do {
    if (!mark) {
        while (r < s) { advance r }
        while (r > s) { advance s }
        // mark start of \block" of S
        mark = s
     }
    if (r == s) {
        result = <r, s>
        advance s
        yield result
    }
    else {
        reset s to mark
        advance r
        mark = NULL
    }
}
```

7.1 An Important Refinement

An important refinement: You can combine the last sorting phase with the merging phase, provided you have enough room in memory to allocate a page for each run of `[R]` and for each run of `[S]`. The final merge pass is where you allocate a page for each run of `R` and each run of `S`. In this process, you save $2 * ([R] + [S])$ I/Os

To perform the optimization, we must

(1) sort `[R]` and `[S]` (individually, using the full buffer pool for each of them) until you have them both "almost-sorted"; that is, for each table T in {`R`, `S`}, keep merging runs of `T` until you get to the penultimate step, where one more sorting pass would result in a sorted table `T`.

(2) See how many runs of `[R]` and how many runs of `[S]` are left; sum them up. Allocate one page in memory for each run. If you have enough input buffers left to accommodate this (i.e. if runs(R) + runs(S) $\leq B - 1$), then you may use the optimization and you could then save $2 * ([R] + [S])$ I/Os.

(3) If you could not do the optimization described in the previous step, it means that runs(R) + runs(S) $\geq$ B. In the ideal case, the optimization allows us to avoid doing an extra read of both R and S, but this is not possible here, as we don't have an available buffer for each run of R and S.

However, we're not out of options yet! If we can't avoid extra reads for both R and S, perhaps we can avoid an extra read for at least one of them. Because we are trying to minimize I/Os, we would like to avoid the extra read for the larger table R. Could it help us to maybe completely sort just the smaller table S and use that in an optimization?

It turns out that we can! If we completely sort S, there is, by definition, now only one sorted run for that table, and this would mean that we only need to allocate one page in our buffer for it. So, if runs(R) + 1 $\leq B - 1$, then we can allocate one buffer page for S and for each run of R. Then, we can perform the SMJ optimization. Here, we have avoided the final sorting pass for R by combining it with the join phase. So we have saved $2*[R]$ pages.

Sometimes, though, even this is not enough – if runs(R) = B-1, then we don't have a spare buffer page for S. In this case, we'd like to still reduce as many I/Os as possible, so maybe perform the procedure described in the previous paragraph, but for the *smaller* table. That is, if runs(S) + 1 $\leq B - 1$, then sort R, reducing the number of sorted runs for it to 1 by definition, and then allocate one buffer page for R and one for each run of S. Then, perform the optimization – this will enable us to avoid the final sorting pass for S by combining it with the join phase, saving us $2 * [S]$ I/Os.

If none of those conditions are met, though, we just won't be able to optimize sort-merge join. And that's alright. Sometimes, we just have to bite the bullet and accept that we can't always cut corners.

8 Practice Questions

1. We have a table R with 100 pages and S with 50 pages and a buffer of size 12. What is the cost of a page nested loop join of R and S?
2. Given the same tables, R and S, from the previous question, we now also have an index on table R on the column a. If we are joining `R.b == S.b`, can we use index nested loop join?
3. Given the same tables, R and S, we want to join R and S on `R.b == S.b`. What is the cost of an index nested loop join of R and S?

   Assume the following for this problem only:

   - $\rho_R = 10$ and $\rho_S = 30$
   - For every tuple in R, there are 5 tuples in S that satisfy the join condition and for every tuple in S, there are 20 tuples in R that satisfy the join condition.
   - There is an Alt. 3 clustered index on R.b of height 3.
   - There is an Alt. 3 unclustered index on S.b of height 2.
   - There are only 3 buffer pages available.
4. Then we realize that R is already sorted on column b so we decide to attempt a sort merge join. What is the cost of the sort merge join of R and S?
5. Lastly, we try Grace Hash Join on the two tables. Assume that the hash uniformly distributes the data for both tables. What is the cost of Grace Hash Join of R and S?

Iterators and Joins

9 Solutions

1. It does not matter the size of the buffer because we are doing a page nested loop join which only requires 3 buffer pages (one for R, one for S, and one for output). We will consider both join orders:

$$[R] + [R][S] = 100 + 100 * 50 = 5100$$

$$[S] + [S][R] = 50 + 50 * 100 = 5050$$

   The second join order is optimal and the number of I/O's is 5050.

2. The answer is no, we cannot use index nested loop join because the index is on the column a which is not going to help us since we are joining on B.

3. Recall the generic formula for an index nested loop join: $[R] + |R|*$(cost to look up matching records in `S`).

   Let's first compute the cost of R join S.

   We know that $|R| = [R] * \rho_R = 100 * 10 = 1000$ records.
   The cost to lookup a record in the unclustered index on S.b of height 2 will cost: 3 I/Os to read the root to the leaf node and 1 more I/O to read the actual tuple from the data page. Since we have 5 tuples in S that match every tuple in R and the S.b index is unclustered the cost to find the matching records in S will be $3 + 5 = 8$ I/Os. Also, because the index is Alt. 3, we know all matching (key, rid) entries will be stored on the same leaf node, so we only need to read in 1 leaf node.

   Therefore, R INLJ S costs $100 + 1000 * 8 = 8100$ I/Os.

   We now compute the cost of S join R.

   We know that $|S| = [S] * \rho_S = 50 * 30 = 1500$ records.
   The cost to lookup a record in the clustered index on R.b of height 3 will cost: 4 I/Os to read the root to the leaf node and 1 I/O for every page of matching tuples. Since there are 20 tuples in R that satisfy the join condition for each tuple in S and $\rho_R = 10$, each index lookup will result in 2 pages of matching tuples, which costs 2 I/Os to read. Thus, the cost to find the matching records in R will be $4 + 2 = 6$ I/Os. Also, because the index is Alt. 3, we know all matching (key, rid) entries will be stored on the same leaf node, so we only need to read in 1 leaf node.

   Therefore, S INLJ R costs $50 + 1500 * 6 = 9050$ I/Os.

   The first join order is optimal so the cost of the INLJ is 8100 I/Os.

4. Since R is already sorted, we just have to sort S which will take 2 passes. And then we merge the tables together using the merge pass. The total cost is therefore $2 * 2 * [S] + [R] + [S] = 4 * 50 + 100 + 50 = 350$

5. First, we need to partition both tables into partitions of size $B-2$ or smaller.

   For R: $ceil(100/11) = 10$ pages per partition after first pass

   For S: $ceil(50/11) = 5$ pages per partition after first pass

   As we see from above, each table only needs one pass to be partitioned. We will have 100 I/O's for reading in table R and then $11 * 10 = 110$ I/O's to write the partitins of table R back to disk. Similar calculations are done for table S.

   Then we join the corresponding partitions together. To do this, we will have to read in 11 partitions of table R and 11 partitions of table S. This gives us $11*10 = 110$ I/O's for reading partitions of table R and $11 * 5 = 55$ I/O's for reading partitions of table S.

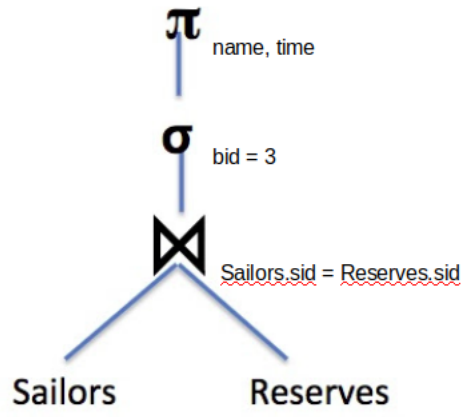
   Therefore, our total I/O cost will be $100 + 110 + 50 + 55 + 110 + 55 = 480$.

CS 186
Spring 2021

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 I/Os. Query Optimization is all about finding the **query plan** that minimizes the number of I/Os 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 I/Os 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. All values in the following examples are integers, as floats have different formulas.

- **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) + 1)
- **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 $|A| * |B|$ tuples 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 tuples we can expect to move on is:

$$\frac{|A|*|B|}{max(unique\ vals\ for\ A.id,\ unique\ vals\ for\ B.id)}$$

4 Common Heuristics

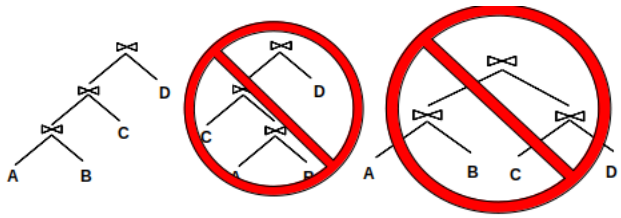
There are way too many possible query plans for a reasonably complex query to analyze them all. We need some way to cut down the number of plans that we actually consider. For this reason, we will use a few heuristics:

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 can pass the pages up one at a time to the next join operator – we don't actually have to write the result of a join to disk.

The third heuristic is beneficial because cross joins produce a ton of pages which makes the operators above the cross join perform many I/Os. 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 I/Os required for each type of scan will not be the same, however. For a table

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

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

(cost to reach level above leaf) + (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] I/Os.

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

(cost to reach level above leaf) + (num of leaf nodes read) + (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 I/Os 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 I/Os 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 I/Os) 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 I/Os 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 I/Os)
2. Index Scan players.age (90 I/Os)
3. Index Scan players.teamid (120 I/Os)
4. Full Scan teams (300 I/Os)
5. Index Scan teams.record (400 I/Os)

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 BNLJ 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 BNLJ B (estimated cost: 1000)
2. B BNLJ A (estimated cost: 1500)
3. A SMJ B (estimated cost: 2000)
4. B BNLJ C (estimated cost: 800)
5. C BNLJ 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} BNLJ C (estimated cost: 10,000)
2. Join 1 {A, B} SMJ C (estimated cost: 12,000)
3. Join 5 {B, C} BNLJ A (estimated cost 8,000)
4. Join 5 {B, C} SMJ A (estimated cost: 20,000)
5. Join 6 {B, C} BNLJ 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 is not interesting to us.

7 Calculating I/O Costs for Join Operations

When we are estimating the I/O costs of the queries we are trying to optimize, it is important to consider the following:

1. Whether we materialize intermediate relations (outputs from previous operators) or stream them into the input of the next operator.
2. If interesting orders obtained from previous operators may reduce the I/O cost of performing a join.

Due to these considerations, we may not be able to directly use the formulas from the Iterators & Joins module to calculate I/O cost.

7.1 Consideration 1

Whether we materialize intermediate relations (outputs from previous operators) or stream them into the input of the next operator can affect the I/O cost of our query.

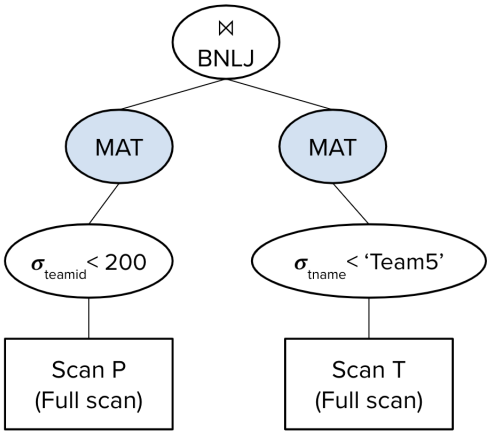
To materialize intermediate relations, we write the output of an intermediate operator to disk (i.e. writing it to a temporary file), a process that incurs additional I/Os. When this data is passed onto the next operator, we must read it from disk into memory again. On the other hand, if we stream the output of one operator into the next, we do not write the intermediate outputs to disk. Instead, as soon as tuples become available when we process the first operator, they remain in memory and we begin applying the second operator to them.

For example, when we push down selections/projections, our goal is to reduce the size of our data so that fewer I/Os are necessary for reading in this data as we traverse up the tree. However, we will only see an impact in I/O cost if the outputs of these selections/projections are materialized, or if the next operator we perform only makes a single pass through the data (i.e. left relations of S/P/BNLJ, left relation of INLJ, etc.). To illustrate this, let's look at the following example.

Example:
Assume B = 5, [P] = 50, [T] = 100, there are 30 pages of players records where P.teamid > 200, and there are 40 pages worth of records where T.tname > 'Team5'. We are optimizing the following query:

```
SELECT *
FROM players P INNER JOIN teams T
ON P.teamid = T.teamid
WHERE P.teamid > 200 AND T.tname > 'Team5';
```

Suppose our query optimizer chose the following query plan:

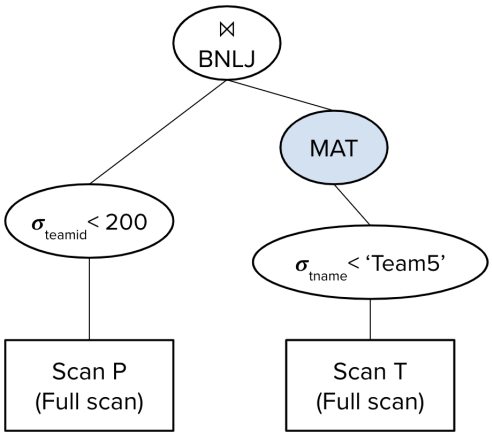


The estimated I/O cost of this query plan would be $(50+30)+(100+40)+(30+\left\lceil\frac{30}{5-2}\right\rceil*40)=650$ I/Os.

We first perform a full scan on P: we read in all 50 pages of P, identify the records where P.teamid >200, and write the 30 pages of those records to disk. This happens because we are materializing the output we get after the selection. We repeat the same process for T: we read in all 100 pages of T, identify the records where T.tname > 'Team5', and write the 40 pages of those records to disk.

Next, when we join the two tables together with BNLJ, we are working with the 30 pages of P and 40 pages of T that the selections have been applied to and currently lie on disk. For each block of B-2 pages of P, we read in all of T, hence the $(30+\left\lceil\frac{30}{5-2}\right\rceil*40)$ term.

Now, suppose our query optimizer instead chose the following query plan:

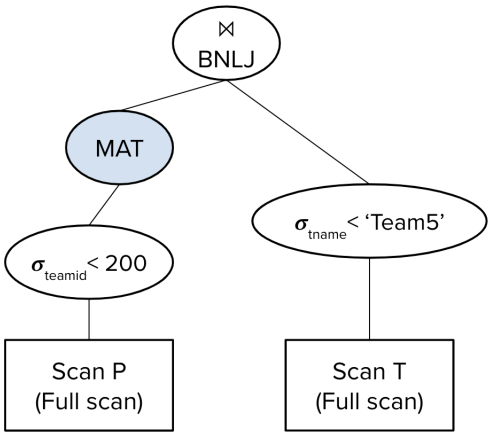


The estimated I/O cost of this query plan would be $(100 + 40) + (50 + \left\lceil \frac{30}{5-2} \right\rceil * 40) = 590$ I/Os.

Since we materialize the output we get after performing the selection on T, we first read in all 100 pages of T, identify the records where T.tname > 'Team5', and write the 40 pages of those records to disk.
Then, we perform a full scan over P, reading in each of the 50 pages into memory. This costs 50 I/Os, and we will filter for records with P.teamid < 200 in memory. Once we've identified a block of B-2 pages of P where P.teamid < 200, we need to find the matching records in T by reading in the 40 pages of T from disk that were materialized after the selection on T.tname < 'Team5'. This process repeats for every block of P, and since we have 30 pages of records where P.teamid < 200 and each block has size B-2, there are approximately $\left\lceil \frac{30}{5-2} \right\rceil$ blocks of P to process.

This is why the cost of the BNLJ is $50 + \left\lceil \frac{30}{5-2} \right\rceil * 40$. This diverges from the BNLJ formula presented in the Iterators & Joins module: we read in all 50 pages of P, but because we push down the selection on P, only 30 pages of P will be passed along to the BNLJ operator and we only need to find matching T records for these pages.

Now, suppose our query optimizer instead chose the following query plan:



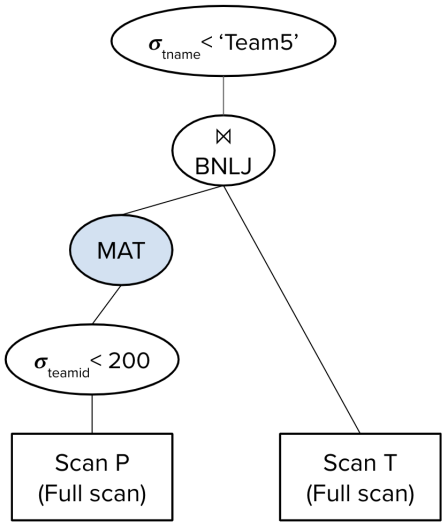
The estimated I/O cost of this query plan would be $(50 + 30) + (30 + \left\lceil \frac{30}{5-2} \right\rceil * 100) = 1110$ I/Os.

Since we materialize the output we get after performing the selection on P, we first read in all 50 pages of P, identify the records where P.teamid > 200, and write the 30 pages of those records to disk.

We will read in the 30 pages of P remaining after the selection in blocks of size B-2. For every block of P, we need to identify the matching records in T. Since we do not materialize after

selecting records where tname < 'Team5', we will have to perform a full scan on the entire T table for every single block of P, reading in T page-by-page into memory, finding records where tname < 'Team5', and joining them with records in P. There are $\left\lceil \frac{30}{5-2} \right\rceil$ blocks of P to process, hence the $\left\lceil \frac{30}{5-2} \right\rceil * 100$ term.

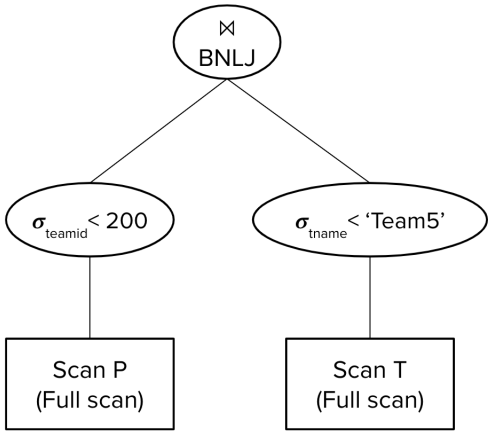
Here, we see that even though we pushed down the selection on T, it did not impact the I/O cost of the query. The I/O cost would be the same if we used this query plan instead:



The reason why the I/O cost remains unaffected is that every time we make a pass through T (once per block of P), we still need to access the entire T table and apply the selection on the fly since we did not save these intermediate results. This shows how pushing down selections/projections affects our I/O cost only when the next operator makes a single pass through the data or when outputs of intermediate relations are materialized.

For the System R query optimizer we are studying in this class, assume that we never materialize the outputs of any intermediate operators and that they are streamed in as the input to the next operator.

Thus, this is the query plan that the System R query optimizer would consider:



The estimated I/O cost of this query plan would be $50 + \left\lceil \frac{30}{5-2} \right\rceil * 100 = 1050$ I/Os.

We perform a full scan over P, reading in each of the 50 pages into memory and filtering for records with P.teamid < 200. Since we push down this selection, only 30 pages of P will be passed onto the BNLJ operator. For every block of B-2 pages of P where P.teamid < 200, we need to identify the matching records in T. Since we do not materialize after selecting records where tname < 'Team5', we will have to perform a full scan on the entire T table for every single block of P, reading in T page-by-page into memory, finding records where tname < 'Team5', and joining them with records in P. There are $\left\lceil \frac{30}{5-2} \right\rceil$ blocks of P to process, hence the $\left\lceil \frac{30}{5-2} \right\rceil * 100$ term.

7.2 Consideration 2

We must also consider the effects of interesting orders from previous operators. For example, let's say we are optimizing the following query, where the players table contains 50 pages and the teams table contains 100 pages:

```
SELECT *
FROM players P INNER JOIN teams T
ON P.teamid = T.teamid
WHERE P.teamid > 200;
```

Suppose we push down the selection P.teamid > 200, choose an index scan on P.teamid costing 60 I/Os as our access plan for the P table, and choose a full scan costing 100 I/Os as our access plan for the T table. The output of the index scan will be sorted on teamid, which is an interesting order since it may be used in the downstream join between P and T.

Now, suppose that we choose SMJ as our join algorithm for joining together P and T. The formula that we presented in the Iterators & Joins module would estimate the I/O cost to be: cost to sort P + cost to sort T + (50 + 100), where the last term ([P] + [T]) is the average case cost of the

merging step and accounts for the cost of reading in both tables.

However, if we use the index scan on P.teamid, we no longer need to run external sorting on P! Instead, we will account for the I/O cost of accessing P using the index scan and stream in the output of this index scan into the merging phase. Thus, the estimated cost of this query plan would be (cost to sort T + 60 + 100) I/Os.

8 Practice Questions

Given the following tables:

```
Teams (teamid, tname, currentcoach)
Players (playerid, pname, teamid, points)
Coaches (coachid, cname)
```

Table	Pages	Column Info	Indexes
Teams	100		
Players	500	Rating has values [1, 10]	Clustered on `rating` (height = 1, 100 leaf pages) Clustered on `playerid` (height = 1, 50 leaf pages)
Coaches	200		

Answer questions 1-5 about the following query:

```
SELECT pname, cname
FROM Teams t
INNER JOIN Players p ON t.teamid = p.teamid
INNER JOIN Coaches c ON t.currentcoach = c.coachid
WHERE p.rating > 5
ORDER BY p.playerid;
```

1) What is the selectivity of the p.rating > 5 condition?

2) What single table access plans will advance to the next pass?

   a Full scan on Teams

b Full scan on Players

c Index scan on Players.rating

d Index scan on Players.playerid

e Full scan on Coaches

3) Which of the following 2 table plans will advance from the second pass to the third pass? Assume that the I/Os provided is optimal for those tables joined in that order and also assume that the output of every plan is not sorted.

a Teams Join Players (1000 I/Os)

b Players Join Teams (1500 I/Os)

c Players Join Coaches (2000 I/Os)

d Coaches Join Players (800 I/Os)

e Teams Join Coaches (500 I/Os)

f Coaches Join Teams (750 I/Os)

4) Which of the following 3 table plans will be considered during the third pass? Again assume that the I/Os provided is optimal for joining the tables in that order and that none of the outputs are sorted.

a (Teams Join Players) Join Coaches - 10,000 I/Os

b Coaches Join (Teams Join Players) - 8,000 I/Os

c (Players Join Teams) Join Coaches - 9,000 I/Os

d (Players Join Coaches) Join Teams - 12,000 I/Os

e (Teams Join Coaches) Join Players - 15,000, I/Os

5) Which 3 table plan will be used?

6) Now, suppose we are trying to optimize the following query using the System R query optimizer:

```
SELECT *
FROM players P INNER JOIN teams T
ON P.teamid = T.teamid
WHERE P.teamid > 200 AND T.tname > 'Team5';
```

Assume B = 5, [P] = 50, [T] = 100, there are 30 pages of players records where P.teamid > 200, and there are 40 pages worth of records where T.tname > 'Team5'.
Also, assume that the access plans we chose in Pass 1 are:

1. Index scan on P.teamid: 60 I/Os
2. Full scan on T: 100 I/Os

Given this information, estimate the average case I/O cost of P SMJ T for Pass 2, assuming we are using unoptimized SMJ.

9 Solutions

1) **1/2**. You can use the formula in lecture/discussion slides, but these are often quicker to solve off the top of your head. Rating can be between 1 and 10 (10 values) and there are 5 values that are greater than 5 (6, 7, 8, 9, 10), so the selectivity is 5/10 or 1/2.

2) **a, c, d, e**. Every table needs at least one access plan to advance, and for Teams and Coaches there is only one option - the full scan. For Players the optimal plan will advance as well as the optimal plan with an interesting order on playerid (playerid is interesting to us because it is used in an ORDER BY later in the query). A full scan takes 500 I/Os. The scan on rating takes 1 + 0.5(100) + 0.5(500) = 301 I/Os. The scan on playerid takes 1 + 50 + 500 = 551 I/Os. The optimal access plan is the index scan on rating, so it advances, and the index scan on playerid is the only plan that produces output sorted on playerid so it advances as well.

3) **a, e**. First recognize that no plan that involves joining Players with Coaches will be considered because there is no join condition between them so it would be a cross join and we don't consider cross joins unless they are our only option for that pass. Then we just pick the best plan for each set of tables which in this case is Teams Join Players for the (teams, players) set and Teams Join Coaches for the (Teams, Coaches) set.

4) **a, e**. B is not considered because it is not a left deep join. C and D both attempt to join 2 table plans that did not advance from pass 2 with a single table plan. A and E (the only correct answers) correctly join a 2 table plan from pass 2 with a single table plan and they are both left deep plans.

5) **(Teams Join Players) Join Coaches**. It has the fewest I/Os out of the plans we are considering. There is no potential for interesting orders because none of the results are sorted, so only one plan can advance.

6) **((100 + 40) + (40+40) + (40+40)) + 60 + 40 = 400 I/Os**.

The cost of sorting T by T.teamid (to completion, since we are performing unoptimized SMJ) is (100 + 40) + (40+40) + (40+40) I/Os:

- In Pass 0 of external sorting, we read in 100 pages of T, but apply the selection on T to the tuples in memory and only write sorted runs for the 40 pages of T that satisfy T.tname > 'Team5'. With B=5, we will produce 8 sorted runs of 5 pages each. This costs 100 + 40 I/Os.
- In Pass 1 of external sorting, we are able to merge B-1 = 4 sorted runs together at a time, producing 2 sorted runs. This costs 40 + 40 I/Os.
- In Pass 2 of external sorting, we merge together the 2 sorted runs, producing 1 sorted run containing all our data. This costs 40 + 40 I/Os.

The access plan our query optimizer chose for P in Pass 1 is the index scan on P.teamid. This will produce output that is already sorted on P.teamid, so we don't need to run external sorting on P. When we proceed to the merging phase of SMJ, we will account for the I/O cost of accessing P using this index scan (60 I/Os) and stream in its output to the merging phase. For T, we will read in the 40 pages from the final sorted run on disk produced after external sorting. Thus, the merging phase costs 60 + 40 I/Os in the average case.

Note: If we chose to apply the SMJ optimization, we would be able to achieve an I/O cost of ((100 + 40) + (40+40)) + 60 + 40 = 320 I/Os. Rather than sorting T to completion, we could stop after Pass 1 of external sorting so that we have 2 sorted runs of T that have not yet been merged together. Then, in the merging phase of SMJ, we allocate our 5 buffer pages the following way:

- 1 buffer page is reserved as the output buffer
- Since we are performing the optimization, 2 buffer pages are reserved for reading in the runs of T. The data in the first run is read page-by-page into one buffer page, and the data in the second run is read page-by-page into the other buffer page.
- We have 2 buffer pages left - these are used for the index scan on P. We can perform the index given only 2 buffer pages because if the index is Alternative 1, we will only need 1 buffer page to store the inner/leaf node currently being processed (it can be evicted as soon as we need to read in another node). If the index is Alternative 2 or 3, we will need 2 buffer pages because when scanning through leaf nodes and reading in the records on data pages, we need 1 page to store the data page and 1 page to store the leaf node. We need more than 1 buffer page because if we evict a leaf node and replace it with the data page a record lies on, we will not be able to efficiently find the next records without the leaf node in memory.

  This problem doesn't specify what alternative the index we're using for the index scan is, but either way, 2 buffer pages will be sufficient for performing this index scan.

CS 186
Spring 2021

Transactions and Concurrency I

1 Introduction

In reality, we usually don't have just one person accessing a database at a time. Many users can make requests to a database at a time which can cause concurrency issues. What happens when one user writes and then another user reads from the same resource? What if both users try to write to the same resource? There are several problems we can run into when several users are using the database at the same time if we're not careful:

- **Inconsistent Reads**: A user reads only part of what was updated.
  - User 1 updates Table 1 and then updates Table 2.
  - User 2 reads Table 2 (which User 1 has not updated yet) and then Table 1 (which User 1 already updated) so it reads the database in an intermediate state.
- **Lost Update**: Two users try to update the same record so one of the updates gets lost. For example:
  - User 1 updates a toy's price to be price * 2.
  - User 2 updates a toy's price to be price + 5, blowing away User 1's update.
- **Dirty Reads**: One user reads an update that was never committed.
  - User 1 updates a toy's price but this gets aborted.
  - User 2 reads the update before it was rolled back.

Transactions and Concurrency I

2 Transactions

Our solution to those problems is to define a set of rules and guarantees about operations. We will do this by using **transactions**. A transaction[1] is a sequence of multiple actions that should be executed as a single, logical, atomic unit. Transactions guarantee the **ACID properties** to avoid the problems discussed above:

- **Atomicity**: A transaction ends in two ways: it either **commits** or **aborts**. Atomicity means that either all actions in the Xact happen, or none happen.
- **Consistency**: If the DB starts out consistent, it ends up consistent at the end of the Xact.
- **Isolation**: Execution of each Xact is isolated from that of others. In reality, the DBMS will interleave actions of many Xacts and not execute each in order of one after the other. The DBMS will ensure that each Xact executes as if it ran by itself.
- **Durabilty**: If a Xact commits, its effects persist. The effects of a committed Xact must survive failures.

3 Concurrency Control

In this note we will discuss how to enforce the **isolation** property of transactions (we will learn how the other properties are enforced in the note about recovery). To do this, we will analyze **transaction schedules** which show the order that operations are executed in. These operations include: **Begin**, **Read**, **Write**, **Commit** and **Abort**.

The easiest way to ensure isolation is to run all the operations of one transaction to completion before beginning the operations of next transaction. This is called a **serial schedule**. For example, the following schedule is a serial schedule because $T1$'s operations run completely before $T2$ runs.

[1] We sometimes shorten transaction to Xact.

Transactions and Concurrency I

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

The problem with these schedules, however, is that it is not efficient to wait for an entire transaction to finish before starting another one. Ideally, we want to get the same results as a serial schedule (because we know serial schedules are correct) while also getting the performance benefits of running schedules simultaneously. Basically, we are looking for a schedule that is **equivalent** to a serial schedule. For schedules to be equivalent they must satisfy the following three rules:

1. They involve the same transactions
2. Operations are ordered the same way within the individual transactions
3. They each leave the database in the same state

If we find a schedule whose results are equivalent to a serial schedule, we call the schedule **serializable**. For example, the following schedule is serializable because it is equivalent to the schedule above. You can work through the following schedule and see that resources A and B end up with the same value as the serial schedule above.

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

Now the question is: how do we ensure that two schedules leave the database in the same final state without running through the entire schedule to see what the result is? We can do this by looking for **conflicting operations**. For two operations to conflict they must satisfy the following three rules:

1. The operations are from different transactions
2. Both operations operate on the same resource
3. At least one operation is a write

We then check if the two schedules order every pair of conflicting operations in the same way. If they do, we know for sure that the database will end up in the same final state. When two schedules order their conflicting operations in the same way the schedules are said to be **conflict equivalent**, which is a stronger condition than being equivalent.

Now that we have a way of ensuring that two schedules leave the database in the same final state, we can check if a schedule is conflict equivalent to a serial schedule without running through the entire schedule. We call a schedule that is conflict equivalent to some serial schedule **conflict serializable**. Note: if a schedule S is conflict serializable then it implies that is serializable.[2].

3.1 Conflict Dependency Graph

Now we have a way of checking if a schedule is serializable! We can check if the schedule is conflict equivalent to some serial schedule because conflict serializable implies serializable. We can check

[2]Not all serializable schedules are conflict serializable

conflict serializability by building a **dependency graph**. Dependency graphs have the following structure:

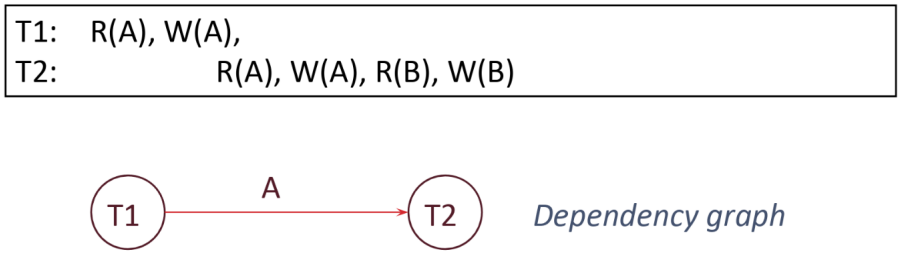
- One node per Xact
- Edge from Ti to Tj if:
  - an operation Oi of Ti conflicts with an operation Oj of Tj
  - Oi appears earlier in the schedule than Oj

A schedule is conflict serializable if and only if its dependency graph is acyclic. So all we have to do is check if the graph is acyclic to know for sure that it is serializable!

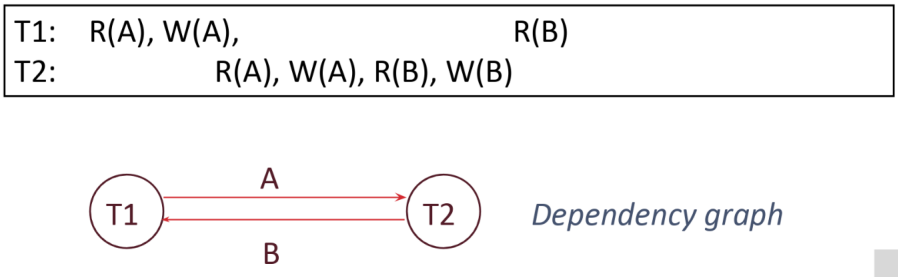
Transactions and Concurrency I

Let's take a look at two examples:

- The following schedule is conflict serializable and the conflict graph is acyclic. There are two conflicting operations:
  - $T1$ reads A and then $T2$ writes to A. Because of this, there will be an edge from $T1$ to $T2$.
  - $T1$ writes to A and then $T2$ reads from A. Since there already is an edge from $T1$ to $T2$, we don't have to add the edge again.



- The following schedule is not conflict serializable and the conflict graph is not acyclic. Some conflicting operations:
  - $T1$ reads A and then $T2$ writes to A. Because of this, there will be an edge from $T1$ to $T2$.
  - $T2$ writes to B and then $T1$ reads B. Because of this, there will be an edge from $T2$ to $T1$.



4 Conclusion

In this note, we removed the naive assumption we have had up until this point of only allowing one user to access a database at a time. We discussed the potential anomalies that can arise if our database does not guarantee the **ACID** properties. We learned how transactions are a powerful mechanism used to encapsulate a sequence of actions that should be executed as a single, logical, atomic unit. In the next note, we will discuss how to actually enforce conflict serializability for our transaction schedules.

CS 186
Spring 2021

Transactions and Concurrency II

1 Introduction

In the last note, we introduced the concept of **isolation** as one of the **ACID properties**. Let's revisit our definition here:

- **Isolation**: Execution of each Xact is isolated from that of others. In reality, the DBMS will interleave actions of many Xacts and not execute each in order of one after the other. The DBMS will ensure that each Xact executes as if it ran by itself.

This note will go into details on how the DBMS is able to interleave the actions of many transactions, while guaranteeing isolation.

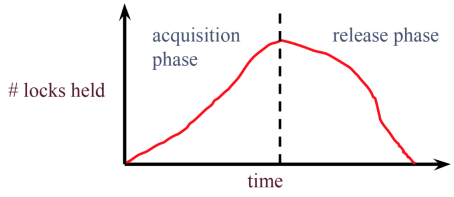
2 Two Phase Locking

What are locks, and why are they useful? Locks are basically what allows a transaction to read and write data. For example, if Transaction $T1$ is reading data from resource A, then it needs to make sure no other transaction is modifying resource A at the same time. So a transaction that wants to read data will ask for a Shared (S) lock on the appropriate resource, and a transaction that wants to write data will ask for an Exclusive (X) lock on the appropriate resource. Only one transaction may hold an exclusive lock on a resource, but many transactions can hold a shared lock on data. **Two phase locking (2PL)** is a scheme that ensures the database uses conflict serializable schedules. The two rules for 2PL are:

- Transactions must acquire a S (shared) lock before reading, and an X (exclusive) lock before writing.
- Transactions cannot acquire new locks after releasing any locks – this is the key to enforcing serializability through locking!

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

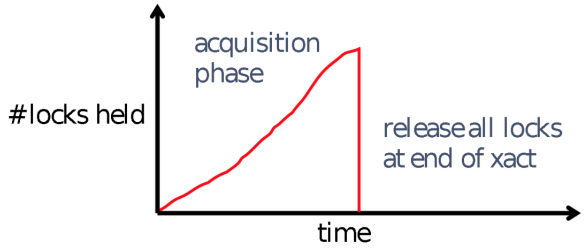


The problem with this is that it does not prevent **cascading aborts**. For example,

- $T1$ updates resource A and then releases the lock on A.
- $T2$ reads from A.

- $T1$ aborts.
- In this case, $T2$ must also abort because it read an uncommitted value of A.

To solve this, we will use **Strict Two Phase Locking**. Strict 2PL is the same as 2PL, except all locks get released together when the transaction completes.



3 Lock Management

Now we know what locks are used for and the types of locks. We will take a look at how the Lock Manager[1] manages these lock and unlock (or acquire and release) requests and how it decides when to grant the lock.

The LM maintains a hash table, keyed on names of the resources being locked. Each entry contains a granted set (a set of granted locks/the transactions holding the locks for each resource), lock type (S or X or types we haven't yet introduced), and a wait queue (queue of lock requests that cannot yet be satisfied because they conflict with the locks that have already been granted). See the following graphic:

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) T4(X)
B	{T6}	X	T5(X) T7(S)

When a lock request arrives, the Lock Manager checks if any Xact in the Granted Set or in the Wait Queue want a conflicting lock. If so, the requester gets put into the Wait Queue. If not, then the requester is granted the lock and put into the Granted Set.

In addition, Xacts can request a lock upgrade: this is when a Xact with shared lock can request to upgrade to exclusive. The Lock Manager will add this upgrade request at the front of the queue.

[1] We will refer to the Lock Manager as LM sometimes.

Here is some pseudocode for how to process the queue; note that it doesn't explicitly go over what to do in cases of promotion etc, but it's a good overview nevertheless.

```
# If queue skipping is not allowed, here is how to process the queue

H = set of held locks on A
Q = queue of lock requests for A

def request(lock_request):
    if Q is empty and lock_request is compatible with all locks in H:
        grant(lock_request)
    else:
        addToQueue(lock_request)

def release_procedure(lock_to_release):
    release(lock_to_release)
    for lock_request in Q:        # iterate through the lock requests in order
        if lock_request is compatible with all locks in H:
            grant(lock_request)   # grant the lock, updating the held set
        else:
            return
```

Note that this implementation does not allow **queue skipping**. When a request arrives under a queue skipping implementation, we first check if you can grant the lock based on what locks are held on the resource; if the lock cannot be granted, then put it at the back of the queue. When a lock is released and the queue is processed, grant *any* locks that are compatible with what is currently held.

For an example of queue skipping and pseudocode, see the appendix. It relies on you understanding multigranulariy locking however, so make sure to read section 7 first to understand the example.

Transactions and Concurrency II

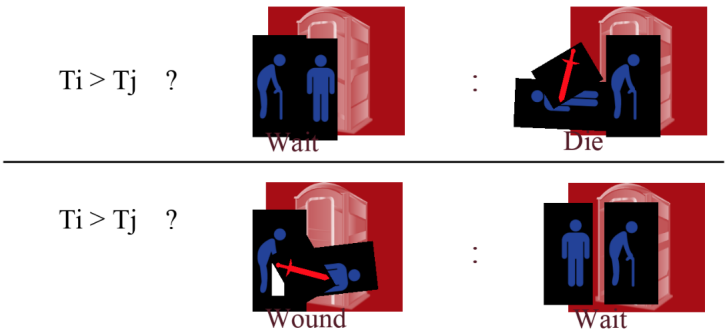
4 Deadlock

We now have a lock manager that will put requesters into the Wait Queue if there are conflicting locks. But what happens if $T1$ and $T2$ both hold S locks on a resource and they both try upgrade to X? $T1$ will wait for $T2$ to release the S lock so that it can get an X lock while $T2$ will wait for $T1$ to release the S it can get an X lock. At this point, neither transaction will be able to get the X lock because they're waiting on each other! This is called a **deadlock**, a cycle of Xacts waiting for locks to be released by each other.

4.1 Avoidance

One way we can get around deadlocks is by trying to **avoid** getting into a deadlock. We will assign the Xact's **priority** by its age: now - start time. If Ti wants a lock that Tj holds, we have two options:[2]

- **Wait-Die**: If Ti has higher priority, Ti waits for Tj; else Ti aborts
- **Wound-Wait**: If Ti has higher priority, Tj aborts; else Ti waits



[2]Important Detail: If a transaction re-starts, make sure it gets its original timestamp.

4.2 Detection

Although we avoid deadlocks in the method above, we end up aborting many transactions! We can instead try detecting deadlocks and then if we find a deadlock, we abort one of the transactions in the deadlock so the other transactions can continue.

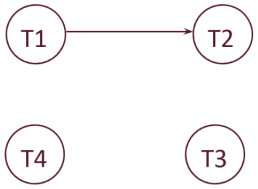
We will detect deadlocks by creating and maintaining a **"waits-for" graph**. This graph will have one node per Xact and an edge from T_i to T_j if:

- T_j holds a lock on resource X
- T_i tries to acquire a lock on resource X, but T_j must release its lock on resource X before T_i can acquire its desired lock.

For example, the following graph has a edge from $T1$ to $T2$ because after $T2$ acquires a lock on B, $T1$ tries to acquire a conflicting lock on it. Thus, $T1$ waits for $T2$.

Example:

T1: S(A) S(D) S(B)
T2: X(B)
T3:
T4:



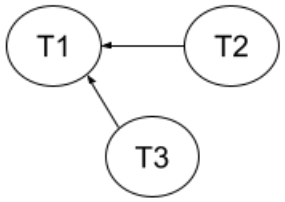
If a transaction T_i is waiting on another transaction T_j (i.e. there is an edge from T_i to T_j), then T_i cannot acquire any new locks. Therefore, a transaction T_k will not wait for T_i on a resource X unless T_i had acquired a conflicting lock on X **before** it began waiting for T_j.

Consider the example below, while keeping in mind that only lock acquisitions are shown in schedule, not lock releases.

Transactions and Concurrency II

Example:

T1: X(A)
T2: S(A) X(B)
T3: S(B) X(A)



There is an edge from $T2$ to $T1$ because $T1$ holds an X lock, when $T2$ requests a conflicting S lock on resource A. Once $T2$ waits for $T1$ to finish with resource A, none of $T2$'s operations can proceed until it is removed from the wait queue. This is why $T3$ does not wait for $T2$ when acquiring an S lock on B, since $T2$ was never actually able to acquire an X lock on B, as it was still waiting on $T1$. Similarly, when $T3$ goes to acquire an X lock on A, it need only wait for $T1$ since at that point in time the only transaction with a conflicting lock on A is $T1$. Note that at that point both $T2$ and $T3$ will be in the wait queue for resource A.

We will periodically check for cycles in a graph, which indicate a deadlock. If a cycle is found, we will "shoot" a Xact in the cycle and abort it to break the cycle.

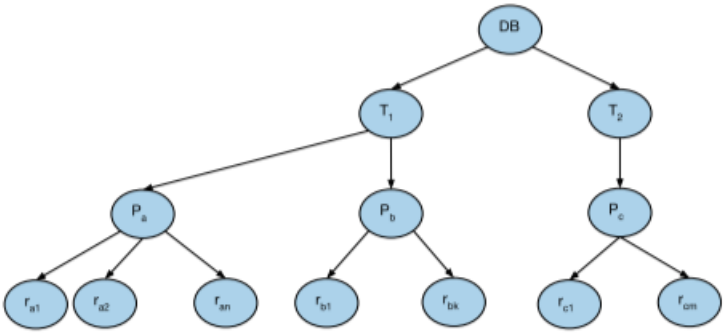
Important note: A "waits-for" graph is used for cycle detection and is different from the conflict dependency graph we discussed earlier (in the previous note) which was used to figure out if a transaction schedule was serializable.

Transactions and Concurrency II

5 Lock Granularity

So now that we understand the concept of locking, we want to figure out what to actually lock. Do we want to lock the tuple containing the data we wish to write? Or the page? Or the table? Or maybe even the entire database, so that no transaction can write to this database while we're working on it? As you can guess, the decision we make will differ greatly based upon the situation we find ourselves in.

Let us think of the database system as the tree below:



The top level is the database. The next level is the table, which is followed by the pages of the table. Finally, the records of the table themselves are the lowest level in the tree.

Remember that when we place a lock on a node, we implicitly lock all of its children as well (intuitively, think of it like this: if you place a lock on a page, then you're implicitly placing a lock on all the records and preventing anyone else from modifying it). So you can see how we'd like to be able to specify to the database system exactly which level we'd really like to place the lock on. That's why multigranularity locking is important; it allows us to place locks at different levels of the tree.

We will have the following new lock modes:

- IS: Intent to get S lock(s) at finer granularity.
- IX: Intent to get X lock(s) at finer granularity. Note: that two transactions can place an IX lock on the same resource – they do not directly conflict at that point because they could place the X lock on two different children! So we leave it up to the database manager to ensure that they don't place X locks on the same node later on while allowing two IX locks on the same resource.

- SIX: Like S and IX at the same time. This is useful if we want to prevent any other transaction from modifying a lower resource but want to allow them to read a lower level. Here, we say that at this level, I claim a shared lock; now, no other transaction can claim an exclusive lock on anything in this sub-tree (however, it can possibly claim a shared lock on something that is not being modified by this transaction–i.e something we won't place the X lock on. That's left for the database system to handle).

Interestingly, note that no other transaction can claim an S lock on the node that has a SIX lock, because that would place a shared lock on the entire tree by two transactions, and that would prevent us from modifying anything in this sub-tree. The only lock compatible with SIX is IS.

Here is the compatibility matrix below; interpret the axes as being transaction $T1$ and transaction $T2$. As an example, consider the entry X, S – this means that it is not possible for $T1$ to hold an X lock on a resource while $T2$ holds an S lock on the same resource. NL stands for no lock.

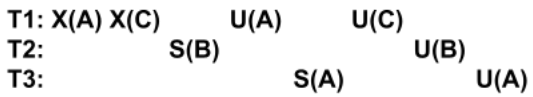
Mode	NL	IS	IX	S	SIX	X
NL	Yes	Yes	Yes	Yes	Yes	Yes
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	Yes	No	No	No
S	Yes	Yes	No	Yes	No	No
SIX	Yes	Yes	No	No	No	No
X	Yes	No	No	No	No	No

5.1 Multiple Granularity Locking Protocol

1. Each Xact starts from the root of the hierarchy.
2. To get S or IS lock on a node, must hold IS or IX on parent node.
3. To get X or IX on a node, must hold IX or SIX on parent node.
4. Must release locks in bottom-up order.
5. 2-phase and lock compatibility matrix rules enforced as well
6. Protocol is correct in that it is equivalent to directly setting locks at leaf levels of the hierarchy.

6 Practice Problems

1. Is the following schedule possible under 2PL? S means acquiring a shared lock, X means acquiring an exclusive lock, and U means releasing a lock.



2. Is the above schedule possible under strict 2PL?

3. For the schedule below, which (if any) transactions will wait under a "wait-die" deadlock avoidance strategy? The priorities in descending order are: T1, T2, T3, T4.[3]

	1	2	3	4	5	6	7
T1	S(A)						X(C)
T2		X(A)		X(B)			
T3					X(B)		
T4			S(B)			S(C)	

4. For the schedule above, which (if any) transactions will wait under a "wound-wait" deadlock avoidance strategy? The priorities in descending order are: T1, T2, T3, T4.

5. What does the "waits-for" graph look like for the above schedule from problem 3? Is there deadlock?

7 Solutions

1. Yes, the schedule is possible under 2PL, because no transaction acquires a lock after it begins to release locks.

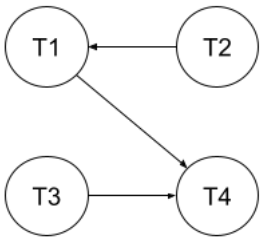
[3]Here the priorities were provided explicitly, but if they are not explicit then you should default to its **age: now - start time**, as defined in 4.1. For this schedule the default priorities in descending order would be: T1, T2, T4, T3 (since T4 began before T3).

2. No, the schedule is not possible under strict 2PL, because $T1$ does not release all of its locks at once. Instead, $T3$ is able to acquire a lock on A after $T1$ releases the X lock on A, but before $T1$ releases the X lock on C. Therefore, the schedule violates strict 2PL since $T3$ could potentially abort under a cascading abort.

3. $T1$ and $T3$
   TS refers to timestep (the top row in the schedule).
   $T2$ will abort at TS-2 since $T2$ has lower priority than $T1$. $T3$ will wait for $T4$ at TS-5 since $T3$ has higher priority than $T4$. $T1$ will wait for $T4$ at TS-7 since $T1$ has higher priority than $T4$.

4. $T2$
   $T2$ will wait for $T1$ at TS-2 since $T2$ has lower priority than $T1$. $T4$ will abort at TS-5 since $T3$ has higher priority than $T4$.

5. There is no deadlock, because there is no cycle in the waits-for graph.



   There is an edge from $T2$ to $T1$ since $T2$ waits for $T1$ at TS-2. This means there is no edge from $T2$ to $T4$ at TS-4 since $T2$ is already waiting for another transaction. There is an edge from $T3$ to $T4$ at TS-5. There is also an edge from $T1$ to $T4$ at TS-7.

Transactions and Concurrency II

Appendix

We now provide a formal proof for why the presence of a cycle in the waits-for graph is equivalent to the presence of a deadlock.

We use $\alpha_j(R_i)$ to represent the lock *request* of lock type α_j on the resource R_i by transaction T_j.

We use $\beta_{ij}(R_i)$ to represent a lock *held* of the lock type β_{ij} on the resource R_i by transaction T_j.

Definition 1. Deadlock

A deadlock is a sequence of transactions (with no repetitions) $T_1, \ldots, T_k$ such that:

- for each $i \in [1, k)$, T_i is requesting a lock $\alpha_i(R_i)$, T_{i+1} holds the lock $\beta_{i,i+1}(R_i)$, and α_i and $\beta_{i,i+1}$ are incompatible, and
- T_k is requesting a lock $\alpha_k(R_k)$, T_1 holds the lock $\beta_{k,1}(R_k)$, and α_k and $\beta_{k,1}$ are incompatible.

Definition 2. Waits-for Graph

Let $T = \{T_1, \ldots, T_n\}$ be the set of transactions and let $D_i \subseteq T$ be defined as follows:

- if T_i is blocked while requesting some lock $\alpha_i(R_i)$, then D_i is the set of transactions T_j that hold locks $\beta_{ij}(R_i)$ where α_i and β_{ij} are incompatible,
- otherwise, $D_i = \emptyset$.

The waits-for graph is the directed graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ and $E = \{(i, j) : T_j \in D_i\}$.

Theorem. There is a simple cycle in the waits-for graph $G \iff$ there is a deadlock.

Proof. Assume there is a simple cycle $C = \{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\} \subseteq E$.

By definition of the waits-for graph, $(i, j) \in E \iff T_j \in D_i$, or alternatively, that T_j holds a lock $\beta_{ij}(R_i)$ while T_i is blocked requesting $\alpha_i(R_i)$, and α_i and β_{ij} are incompatible.

Therefore, $(i_j, i_{j+1}) \in C \subseteq E \iff T_{i_{j+1}}$ holds a lock $\beta_{i_j i_{j+1}}(R_{i_j})$ while T_{i_j} is blocked requesting $\alpha_{i_j}(R_{i_j})$, where α_{i_j} and $\beta_{i_j i_{j+1}}$ are incompatible. A similar result holds for (i_k, i_1).

But this is simply the definition of a deadlock on the transactions $T_{i_1}, \ldots, T_{i_k}$, so we have our result. □

Queue Skipping
An example of queue skipping is the following: Suppose, on resource A, that T_1 holds IS and T_2 holds an IX lock. The queue has, in order, the following requests: $T_3 : X(A)$, $T_4 : S(A)$, $T_5 : S(A)$, and $T_6 : SIX(A)$.

Now, let T_2 release its lock. Instead of processing the queue in order and stopping when a conflicting lock is requested (which would result in no locks being granted, as T_3 is at the front and wants $X(A)$), queue skipping processes the queue in order, *granting locks one by one whenever compatible.*

Here, it would look at T_3's X(A) request, determine that X(A) is incompatible with the IS(A) lock T_1 holds, and move to the next element in the queue. It would then grant T_4's S(A) request, as it is compatible with the held locks of A, and add $T_4 : S(A)$ to the set of locks held on A. It would then look at $T_5 : S(A)$, determine that it is compatible with $T_4 : S(A)$ and $T_1 : IS(A)$, and grant it. Finally, it would look at $T_6 : SIX(A)$, see that it is incompatible with $T_4 : S(A)$ and $T_5 : S(A)$ in the held set, and *not* grant it as a result.

Here is some pseudocode for processing the queue, but this time with queue skipping:

```
# If queue skipping is allowed, here is how to process the queue
H = set of held locks on A
Q = queue of lock requests for A

def request(lock_request):
    if lock_request is compatible with all locks in H:
        grant(lock_request)
    else:
        addToQueue(lock_request)

def release_procedure(lock_to_release):
    release(lock_to_release)
    for lock_request in Q:        # iterate through the lock requests in order
        if lock_request is compatible with all locks in H:
            grant(lock_request)   # grant the lock, updating the held set
```

CS 186
Spring 2021

Recovery

1 Motivation

In prior modules we discussed the ACID properties of transactions. In this note we will discuss how to make our database resilient to failures. The two ACID properties that we will learn how to enforce in this note are:

1. **Durability**: If a transaction finishes (commits), we will never lose the result of the transaction.
2. **Atomicity**: Either all or none of the operations in the transaction will persist. This means that we will never leave the database in an intermediate state. An example of this is swapping a class in CalCentral. There are two operations: dropping your old class and adding the new one. If the database crashes before it can add your new class, you do not actually want to be dropped out of your old class.

2 Force/No Force

Durability can be a very simple property to ensure if we use a force policy. The **force policy** states when a transaction finishes, force all modified data pages to disk before the transaction commits. This would ensure durability because disk is persistent[1]; in other words, once a page makes it to disk it is saved permanently. The downside of this approach is performance. We end up doing a lot of unnecessary writes. The policy we would prefer is **no force** which says to only write back to disk when the page needs to be evicted from the buffer pool. While this helps reduce unnecessary writes, it complicates durability because if the database crashes after the transaction commits, some pages may not have been written to disk and are consequently lost from memory, since memory is volatile. To address this problem, we will redo certain operations during recovery.

3 Steal/No-Steal

Similarly, it would be easy to ensure atomicity with a no-steal policy. The **no-steal policy** states that pages cannot be evicted from memory (and thus written to disk) until the transaction commits. This ensures that we do not leave the database in an intermediate state because if the transaction does not finish, then none of its changes are actually written to disk and saved. The problem with this policy is that it handcuffs how we can use memory. We have to keep every modified page in memory until a transaction completes. We would much rather have a **steal policy**, which allows modified pages to be written to disk before a transaction finishes. This will complicate enforcing atomicity, but we will fix this problem by undoing bad operations during recovery.

[1] When a machine crashes, the bits in memory are "erased," which is why memory is not persistent and we have to rely on persistent storage devices such as disk to guarantee durability.

4 Steal, No-Force

To review, we chose to use two policies (steal, no force) that make it difficult to guarantee atomicity and durability, but get us the best performance. The rest of this note will cover how to ensure atomicity and durability while using a **steal, no force** policy.

5 Write Ahead Logging

To solve these complications we will use logging. A log is a sequence of log records that describe the operations that the database has done.

5.1 Update Log Record

Each write operation (SQL insert/delete/update) will get its own log UPDATE record.

An UPDATE log record looks like this:

<XID, pageID, offset, length, old_data, new_data>

The fields are:

- XID: transaction ID - tells us which transaction did this operation
- pageID: what page has been modified
- offset: where on the page the data started changing (typically in bytes)
- length: how much data was changed (typically in bytes)
- old_data: what the data was originally (used for undo operations)
- new_data: what the data has been updated to (used for redo operations)

5.2 Other Log Records

There are a few other record types we will use in our log. We will add fields to these log records throughout the note as the need for these fields becomes apparent.

- COMMIT: signifies that a transaction is starting the commit process
- ABORT: signifies that a transaction is starting the aborting process
- END: signifies that a transaction is finished (usually means that it finsihed committing or aborting)

5.3 WAL Requirements

Just like regular data pages, log pages need to be operated on in memory, but need to be written to disk to be stored permanently. **Write Ahead Logging (WAL)** imposes requirements for when we must write the logs to disk. The two rules are as follows:

1. Log records must be written to disk ***before*** the corresponding data page gets written to disk. This is how we will achieve atomicity. The intuition for this is that if a data page is written first and then the database crashes we have no way of undoing the operation because we don't know what operation happened!

2. All log records must be written to disk when a transaction commits. This is how we will achieve durability. The intuition is that we need to persistently track what operations a committed transaction has performed. Otherwise, we would have no idea what operations we need to redo. By writing all the logs to disk, we know exactly which operations we need to redo in the event that the database crashes before the modified data pages are written to disk!

6 WAL Implementation

To implement write ahead logging we're going to add a field to our log records called the **LSN**, which stands for Log Sequence Number. The LSN is a unique increasing number that helps signify the order of the operations (if you see a log record with LSN = 20 then that operation happened after a record with LSN = 10). In this class the LSNs will increase by 10 each time, but this is just a convention. We will also add a **prevLSN** field to each log record which stores the last operation from the *same* transaction (this will be useful for undoing a transaction).

The database will also keep track of the flushedLSN which is stored in RAM. The **flushedLSN** keeps track of the LSN of last log record that has been flushed to disk. When a page is **flushed**, it means that the page has been written to disk; it usually also implies that we evict the page from memory because we don't need it there anymore. The flushedLSN tells us that any log records before it should not be written to disk because they are already there. Log pages are usually appended to the previous log page on disk, so writing the same logs multiple times would mean we are storing duplicate data which would also mess up the continuity of the log.

We will also add a piece of metadata to each data page called the pageLSN. The **pageLSN** stores the LSN of the operation that last modified the page. We will use this to help tell us what operations actually made it to disk and what operations must be redone.

7 Inequality Exercise

Before page i is allowed to be flushed to disk, what inequality must hold?

$$pageLSN_i \_\_ flushedLSN$$

Answer: $\leq$, This comes from our first rule for WAL - we must flush the corresponding log records before we can flush the data page to disk. A data page is only flushed to disk if the LSN of the last operation to modify it is less than or equal to the flushedLSN. In other words, before page i can be flushed to disk, the log records for all operations that have modified page i must have been flushed to disk.

8 Aborting a Transaction

Before getting into recovering from a crash, let's figure out how a database can abort a transaction that is in progress. We may want to abort a transaction because of deadlock, or a user may decide to abort because the transaction is taking too long. Transactions can also be aborted to guarantee the C for **consistency** in ACID if an operation violates some integrity constraint. Finally, a transaction may need to be aborted due to a system crash! We need to ensure that none of the operations are still persisted to disk once the abort process finishes.

8.1 Abort and CLR Log Records

The first thing we will do is write an ABORT record to the log to signify that we are starting the process. Then we will start at the last operation in the log for that transaction. We will undo each operation in the transaction and write a CLR record to the log **for each undone operation**. A **CLR** (Compensation Log Record) is a new type of record signifying that we are undoing a specific operation. It is essentially the same thing as an UPDATE record (it stores the previous state and the new state), but it tells us that this write operation happened due to an abort.

9 Recovery Data Structures

We will keep two tables of state to make the recovery process a little bit easier. The first table is called the **transaction table** and it stores information on the active transactions. The transaction table has three fields:

- XID: transaction ID
- status: either running, committing, or aborting
- **lastLSN**: the LSN of the most recent operation for this transaction

An example of the transaction table is here:

Transaction Table

XID	Status	lastLSN
1	R	33
2	C	42

The other table we maintain is called the **Dirty Page Table (DPT)**. The DPT keeps track of what pages are dirty (recall from many modules ago that dirty means the page has been modified in memory, but has not been flushed to disk yet). This information will be useful because it will tell us what pages have operations that have not yet made it to disk. The DPT only has two columns:

- Page ID
- **recLSN**: the *first* operation to dirty the page

An example of the DPT is here:

Dirty Page Table

PageID	recLSN
46	11
63	24

One thing to note is that both of these tables are stored in memory; so when recovering from a crash, you will have to use the log to reconstruct the tables. We will talk about a way to make this easier (checkpointing) later in the note.

10 More Inequality Questions

1. Fill in the equality below to enforce the WAL rule that all the logs must be flushed to disk before a transaction T can commit:

   $$flushedLSN \_\_ lastLSN_T$$

   Answer: $\geq$ If the flushedLSN is is greater than the last operation of the transaction then we know all of the logs for that transaction are on disk.

2. For a page P that is in the DPT, fill in the following inequality for what must always be true:

   $$recLSN_P \_\_ in\ memory\ pageLSN_P$$

   Answer: $\leq$ If a page is in the dirty page table then it must be dirty, so the last update must not have made it to disk. The recLSN is the *first* operation to dirty the page, so it must be smaller than the last operation to modify that page.

3.

$$recLSN_P \_\_ on\ disk\ pageLSN_P$$

Answer: > If the page is dirty then the operation that dirtied the page (recLSN) must not have made it to disk, so it must have came after the operation that did make it to disk for that page.

11 Undo Logging

We've covered a lot of background information on how the database writes to its log and how it aborts transactions when it is running normally. Now, let's finally get into the reason for all this logging - recovering from failures. One possible recovery mechanism is Undo Logging. Note that Undo Logging does not, in fact, use write ahead logging (WAL) that we previously discussed (we will get back to that in a bit). Further, it uses the force and steal mechanisms in terms of buffer pool management.

The high level idea behind Undo Logging is that we want to undo the effects of all the transactions that have not yet been committed, while not doing so for those that have. In order to do this, we establish 4 types of records: Start, Commit, Abort and Update (which contains old value). We also need to establish 2 rules regarding how we do logging and when to flush dirty data pages to disk:

1. If a transaction modifies a data element, then the corresponding update log record must be written to the disk before the dirty page containing it is written. We want this because we would like to ensure the old value is recorded on the disk before the new value replaces it permanently.

2. If a transaction commits, then the pages that are modified must be written to disk before the commit record itself is written to disk. That rule ensures that all changes that are made by the transaction have been written to the disk before the transaction itself actually commits. This is important because if we see the commit log record in the log, then we will consider the transaction as committed and won't undo its changes during recovery. Notice that this is different from write ahead logging; here, the dirty pages are written to the disk **before** writing the commit record itself to the disk.

Notice that the first rule implements the steal policy, since the dirty pages written to the disk before a transaction commits, and the second rule implements the force policy.

Now that we have established these rules, we can discuss recovery with undo logging. When the system crashes, we first run the recovery manager. We scan the log from the end to determine whether each of the transactions is completed or not. The action that we take based on the log record we encounter is as follows:

1. COMMIT/ABORT T: mark T as completed
2. UPDATE T, X, v: if T is not completed, write X=v to disk, else ignore
3. START T: ignore

And how far do we need to go? All the way to the start, unless we have checkpointing (discussed later).

12 Redo Logging

Now, lets talk about another form of logging based recovery called Redo Logging. Here, Redo Logging implements the no-force no-steal strategy for buffer management. In Redo Logging, we have the same type of log records as Undo Logging, though only difference is for the Update log record, where instead of storing the previous value for particular data elements, we store the new value it is going to write.

The idea behind Redo Logging is similar to Undo Logging, except that at recovery time instead of undoing all the transactions that are incomplete, we redo the actions of all the transactions that were committed instead. Meanwhile, we leave all the uncommitted transactions alone. Like Undo Logging, we also have also have a rule to abide by

1. If a transaction modifies a data element X, then both the update record and commit record must be written to to disk before dirty data page itself - this is the no-steal policy. Hence, the dirty data page is written later than the transaction commit record, and is essentially write ahead logging.

Recovery for Redo Logging is rather straightforward: we just read the log from the beginning, and redo all updates of committed transactions. While this may seem like a lot of operations, it can be optimized, like Undo Logging, through checkpointing.

13 ARIES Recovery Algorithm

When a database crashes, the only things it has access to are the logs that made it to disk and the data pages on disk. From this information, it should restore itself so that all committed transactions' operations have persisted (durability) and all transactions that didn't finish before the crash are properly undone (atomicity). The recovery algorithm consists of 3 phases that execute in the following order:

1. Analysis Phase: reconstructs the Xact Table and the DPT
2. Redo Phase: repeats operations to ensure durability

3. Undo Phase: undoes operations from transactions that were running during the crash to ensure atomicity

Let's go through each phase in detail.

14 Analysis Phase

The entire purpose of the analysis phase is to rebuild what the Xact Table and the DPT looked like at the time of the crash. To do this, we scan through all of the records in the log beginning from the start. We modify our tables according to the following rules:

- On any record that is not an END record: add the transaction to the the Xact Table (if necessary). Set the lastLSN of the transaction to the LSN of the record you are on
- If the record is a COMMIT or an ABORT record, change the status of the transaction in the Xact Table accordingly
- If the record is an UPDATE record, if the page is not in the DPT add the page to the DPT and set recLSN equal to the LSN
- If the record is an END record, remove the transaction from the Xact Table.

At the end of the analysis phase, for any transactions that were committing we will also write the END record to the log and remove the transaction from the Xact Table. Additionally, any transactions that were running at the time of the crash need to be aborted and the abort record should be logged. Note that on several prior exam questions we have asked for the status of the tables before this final pass (without actually saying so on the exam - it was just an assumption from the semester). We promise to be explicit about what we are looking for in the future.

One problem with the analysis phase so far is that it requires the database to scan through the entire log. In production databases this is not realistic as there could be thousands or millions of records. To speed up the analysis phase, we will use **checkpointing**. Checkpointing writes the contents of the Xact Table and the DPT to the log. This way, instead of starting from the beginning of the log, we can start from the last checkpoint. Checkpointing actually writes two records to the log, a $< BEGIN\_CHECKPOINT >$ record that says when checkpointing started and an $< END_CHECKPOINT >$record that says when we finished writing the tables to the log. The tables written to the log can be the state of tables at any point between the $< BEGIN\_CHECKPOINT >$ and $< END\_CHECKPOINT >$. This means we need to start at the $< BEGIN\_CHECKPOINT >$ because we're not sure if the records after it are actually reflected in the tables that were written to the log.

15 Analysis Phase Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction Table

Transaction	Status	lastLSN
T1	running	20
T2	running	30
T3	running	40

Dirty Page Table

Page ID	recLSN
P1	40
P3	10

The database crashed and we are given the log above. The Xact Table and the DPT on the right are the tables found in the $< END\_CHECKPOINT >$ record. Let's go through the process.

First, we start at the record at LSN 60 because it is the record immediately after the begin checkpoint record. This is an UPDATE record and T3 is already in the Xact Table, so we will update the lastLSN to 60. The page it updates is already in the DPT, so we don't have to do anything with the DPT. The tables now look like this:

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Running	60

PageID	recLSN
P1	40
P3	10

Now we go to the record at LSN 70. It is an ABORT record, so we need to change the status in our Xact Table to Aborting and update the lastLSN.

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Aborting	70

PageID	recLSN
P1	40
P3	10

There is nothing to do for the end checkpoint record, so we move onto the CLR (UNDO) at LSN 90. T3 is in the Xact Table so we update the lastLSN and the page it is modifying (P3) is already in the DPT so we again do not have to modify the DPT.

Transaction	Status	lastLSN
T1	Running	20
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10

At LSN 100 we have another update operation and T1 is already in the Xact Table so we will update its lastLSN. The page this record is updating is not in the DPT, however, so we will add it with a recLSN of 100 because this is the first operation to dirty the page.

Transaction	Status	lastLSN
T1	Running	100
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Next is LSN 110 which is a COMMIT record. We need to change T1's status to committing and update the lastLSN.

Transaction	Status	lastLSN
T1	Committing	110
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Finally, LSN 120 is an END record meaning that we need to remove T1 from our Xact Table. This leaves us with a final answer of:

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Note that in this question we left out that final pass for ending committing transactions and aborting running transactions because this has also been done on several exams. In reality, before the Redo Phase starts, we would change T2's status to Aborting.

16 Redo Phase

The next phase in recovery is the Redo Phase which ensures durability. We will repeat history in order to reconstruct the state at the crash. We start at the smallest recLSN in the DPT because that is the first operation that may not have made it to disk. We will redo all UPDATE and CLR operations unless one of the following conditions is met:

- The page is not in the DPT. If the page is not in the DPT it implies that all changes (and thus this one!) have already been flushed to disk.
- recLSN $>$ LSN. This is because the first update that dirtied the page occurred after this operation. This implies that the operation we are currently at has already made it to disk, otherwise it would be the recLSN.
- pageLSN (disk) $\geq$ LSN. If the most recent update to the page that made it to disk occurred after the current operation, then we know the current operation must have made it to disk.

17 Redo Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

The log and final tables from the analysis example have been reproduced for your convenience. Now let's answer the following two questions:

1. Where should we start the recovery process?.
Answer: At LSN 10 because that is the smallest recLSN in the DPT.

2. What are the LSNs of the operations that get redone?
Answer:

- 10 - UPDATE that does not meet any of the conditions
- Not 20 - recLSN > LSN
- Not 30 - page not in DPT
- 40 - UPDATE that does not meet any of the conditions

- Not 50 - only redo UPDATEs and CLRs
- 60 - UPDATE that does not meet any of the conditions
- Not 70 - only redo UPDATEs and CLRs
- Not 80 - only redo UPDATEs and CLRs
- 90 - CLR that does not meet any of the conditions
- 100 - UPDATE that does not meet any of the conditions
- Not 110 - only redo UPDATEs and CLRs
- Not 120 - only redo UPDATEs and CLRs

For a final answer of 10, 40, 60, 90, 100.

18 Undo Phase

The final phase in the recovery process is the undo phase which ensures atomicity. The undo phase will start at the end of the log and works its way towards the start of the log. It undoes every UPDATE (only UPDATEs!) for each transaction that was active (either running or aborting) at the time of the crash so that we do not leave the database in an intermediate state. It will not undo an UPDATE if it has already been undone (and thus a CLR record is already in the log for that UPDATE).

For every UPDATE the undo phase undoes, it will write a corresponding CLR record to the log. CLR records have one additional field that we have not yet introduced called the **undoNextLSN**. The undoNextLSN stores the LSN of the next operation to be undone for that transaction (it comes from the prevLSN of the operation that you are undoing). Once you have undone all the operations for a transaction, write the END record for that transaction to the log.

Appendix 1 explains how this is implemented efficiently.

19 Undo Example

Log

LSN	Record	prevLSN
10	T1 updates P3	null
20	T1 updates P1	10
30	T2 updates P2	null
40	T3 updates P1	null
50	Begin Checkpoint	-
60	T3 updates P3	40
70	T3 Aborts	60
80	End Checkpoint	-
90	CLR undo T3 LSN: 60	70
100	T1 updates P4	20
110	T1 commits	100
120	T1 Ends	110

Transaction	Status	lastLSN
T2	Running	30
T3	Aborting	90

PageID	recLSN
P1	40
P3	10
P4	100

Write down all of the log records that will be written during the undo phase.

Answer: First recognize that the log provided is missing one record from the analysis phase. Remember that at the very end of the analysis phase we need to write log entries for any aborting transactions. Therefore, there should be an ABORT record at LSN 130 that aborts T2. It will have a prevLSN of 30 because that is the last operation that T2 did before this ABORT operation. We include this record in the final answer for completeness, but note that it is not technically written during the Undo Phase, it is written at the end of the analysis phase.

We now move on to undoing the operations for T2 and T3. The most recent update for T3 occurs at LSN 60, but notice that there is already a CLR for that operation in the log (LSN 90). Because that operation is undone, we do not need undo it again.

The next operation is the UPDATE at LSN 40. This update is not undone anywhere else in

the log so we do need to undo it and write the corresponding CLR record. The prevLSN will be 90 because that CLR log record is the last operation for T3. The undoNextLSN will be null because there are no other operations in T3 to undo (see final answer below for the full syntax). Because there are no more actions to undo for T3, we must also write the END record for that transaction.

The next operation we need to undo is the update at LSN 30 for T2. The prevLSN for this record will be 130 because of the ABORT record we wrote before. The undoNextLSN will be null again because there are no other operations for T2 in the log. We will also need to write the END record for T2 because that was last operation we needed to undo. Here is the final answer:

LSN	Record	prevLSN	undoNextLSN
130	ABORT T2	30	
140	CLR Undo T3: LSN 40	90	null
150	END T3	140	
160	CLR Undo T2: LSN 30	130	null
170	END T2	160	

20 Conclusion

In this note we covered how databases can guarantee that they will recover from failure even while using a steal, no-force policy. We covered how the database uses Write Ahead Logging policies to log all operations when it is running correctly. We finally covered how the database uses the log in a 3 step process (analysis, redo, undo) to recover from failure and restore the database to a correct state.

21 Appendix 1: Undo Details

This explanation will rely on the pseudocode from lecture:

```
toUndo = {lastLSNs of all Xacts in the Xact Table}
while !toUndo.empty():
    thisLR = toUndo.find_and_remove_largest_LSN()
    if thisLR.type == CLR:
        if thisLR.undoNextLSN != NULL:
            toUndo.insert(thisLR.undonextLSN)
        else: // thisLR.undonextLSN == NULL
            write an End record for thisLR.xid in the log
    else:
        if thisLR.type == UPDATE:
            write a CLR for the undo in the log
            undo the update in the database
        if thisLR.prevLSN != NULL:
            toUndo.insert(thisLR.prevLSN)
        elif thisLR.prevLSN == NULL:
            write an END record for thisLR.xid
```

The pseudocode uses toUndo which is a max-heap that stores the LSNs of operations that we potentially need to undo. During the undo phase, the only transactions we want to undo are the ones that were still active at the time of the crash. Therefore, we start by adding the lastLSN of all the transactions in the Xact Table (recall that only transactions that are in the Xact Table could have been active at the time of the crash).

We then iterate until the toUndo heap is empty. When toUndo is empty it implies that we have undone everything that we need to. On each iteration, we pop off the record with the largest LSN in the heap. If the record is a CLR record we will add the undoNextLSN to the toUndo heap. The whole purpose of this field is to tell us what record needs to be undone next, so it makes sense to use this during the UNDO process. For any other record, however, we don't have this field so we will need to add the prevLSN to toUndo instead. If the prevLSN field is null (or the undoNextLSN in the case of a CLR), it means we are done with the transaction so we can just write the END

record to the log and we don't have to add anything to toUndo. Remember that UPDATE records are the only records that get undone, so we need the special case for them to write the CLR record to the log and to actually undo the update.

This implementation has a few nice properties. The first is that we always go backwards in the log, we never jump around. This is because the prevLSN/undoNextLSN is always smaller than the LSN of the record and because we always remove the largest LSN from the heap. This is a nice property because it means we will be doing sequential IOs rather than more costly random IOs.

The other nice property is that this implementation allows us to avoid undoing an UPDATE if its already been undone. We have this property because if an UPDATE has been undone, the correspoding CLR will occur after it in the log. Because we go backwards, we will hit the CLR before the UPDATE. We then add the undoNextLSN to toUndo which lets us skip over that original update. This is because undoNextLSN must point to an UPDATE operation before the UPDATE that the CLR undoes, so the next log entry we will read from that transaction will occur before the UPDATE we want to avoid. Because we never go forward during UNDO, it means we will never actually hit that UPDATE.

22 Appendix 2: LSN list

There are a lot of different LSNs, so here is a list of what each one is:

- LSN: stored in each log record. Unique, increasing, ordered identifier for each log record
- flushedLSN: stored in memory, keeps track of the most recent log record written to disk
- pageLSN: LSN of the last operation to update the page (in memory page may have a different pageLSN than the on disk page)
- prevLSN: stored in each log record, the LSN of the previous record written by the current record's transaction
- lastLSN: stored in the Xact Table, the LSN of the most recent log record written by the transaction
- recLSN: stored in the DPT, the log record that first dirtied the page since the last checkpoint
- undoNextLSN: stored in CLR records, the LSN of the next operation we need to undo for the current record's transaction

CS 186
Spring 2021

DB Design

1 Introduction

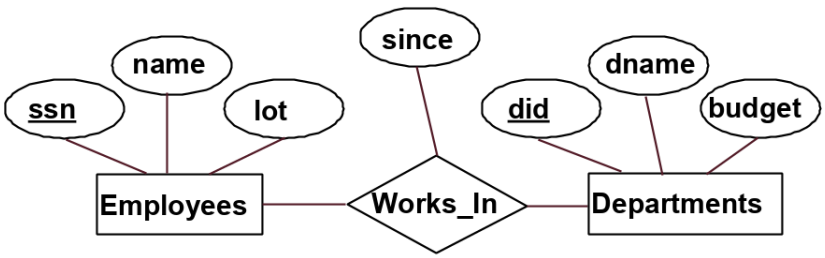
So far, we've learned how to use already built databases: SQL queries! We also learned the internals of a database management system which includes many abstraction levels. But what happens if we are given a high-level description of what data we want to store with our database? How do we design a database to fit out needs? We will discuss these question in this module!

2 Entity-Relationship Models

When designing a database, we often use Entity-Relationship Models (aka "E-R" models). These models have 2 main components.

The first main component is an **entity**: a real-world object described by a set of attribute values. An entity is usually depicted as a rectangle in our E-R model. In addition, the attributes/fields associated with the entity are depicted as ovals. For example, the following figure shows an employee as an entity in our E-R model with the following fields: a SSN number, a name, and a lot. Note that the attribute "ssn" is underlined because it is the identifying attribute for this entity!

The second main component is a **relationship**: association among two or more entities. A relationship is usually depicted as a diamond in our E-R model. Again, the attributes associated with the relationship are depicted as ovals. For example, the following figure shows "works in" as a relationship between the entities employee and department. The "works in" relationship has the attribute "since."

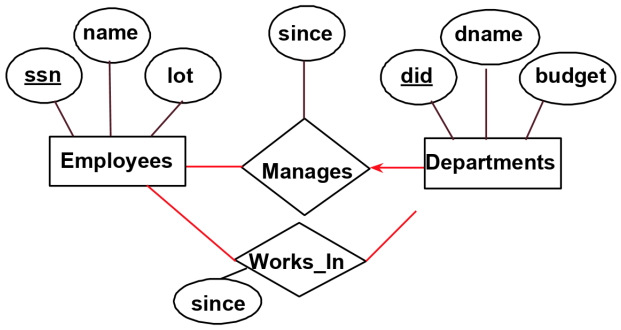


Note that the same entity set can participate in different relationship sets, or in different "roles" in the same relationship set.

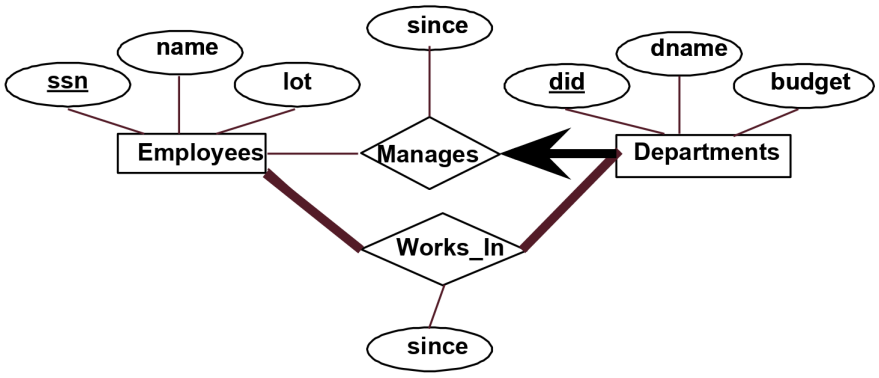
2.1 Relationship Constraints

We have connected our entities to relationships with a thin black line up until now. This line means denotes a **many-to-many** relationship. This means each entity can participate in 0 or more times in the relationship and vice versa. For example, for the figure above, this means that many employees can work in many departments (0 or more) and departments can have many employees (0 or more).

In contrast, we want our model to reflect that each department has at most one manager. We do this by using a **key constraint**, which denotes a **1-to-many** relationship. The key constraint is depicted by a thin arrow. For example, for the following figure, this means that each department has at most one manager (0 or 1) while employees can be managers for many (0 or more) departments.

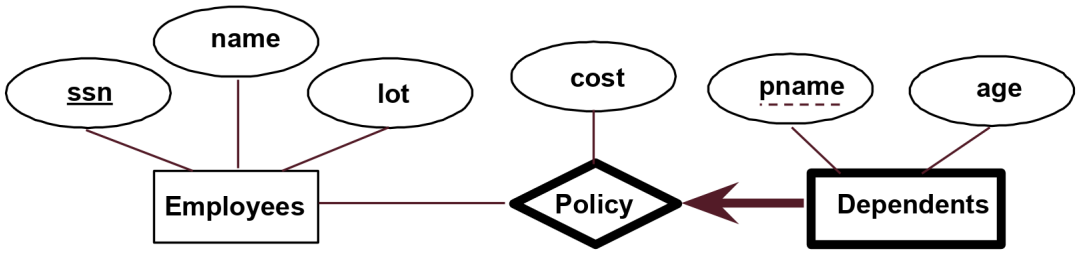


What if we want our model to show that each employee works in at least one department? We do this by using a **participation constraint**, which denotes **at least one** relationship. The participation constraint is depicted by a thick line. For example, in the following figure, we added two thick lines. One of them denotes that each employee works in at least one department. The other denotes that each department has at least one employee.



In addition, you can see we added a thick arrow pointing from departments to manages. This is both a key and participation constraint: each department has at least and at most one manager.

A **weak entity** can be identified uniquely only by considering the primary key of another (owner) entity. Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities). Weak entity set must have total participation in this identifying relationship set. We depict a weak entity by bolding the rectangle and relationship as shown below. In addition, weak entities have only a "partial key" (dashed underline). In the example below, we say that every dependent is uniquely identified by their pname and an employees' ssn and that each employee has 0 or more dependents attached to them.



Now that we have our E-R Model drawn out, how do we actually organize it into relations? We can translate each entity and relationship to its own table and have primary keys and foreign keys depending on the relationship between tables. In general, there are many ways to set up relations to represent the same E-R model.

3 FDs and Normalization

Now that we have set up our database, we want to optimize! One thing we want to avoid is redundancy in our schema. This is wasted storage and also leads to insert/delete/update anomalies. Our solution to this problem is going to be **Functional Dependencies**. A functional dependency $X \rightarrow Y$ means that the X column determines Y column in a table R. This means given any two tuples in table R, if their X values are the same, then their Y values must be the same (but not vice versa).

Let's look at a couple of examples to better understand functional dependencies and what "determines" means. In the example below, two of the rows have the same X value (1) but different Y values (5 and 6), so we cannot say that X determines Y:

X	**Y**
1	5
1	6
2	1

For the following example, we see that all rows with the same X value have the same Y value. More specifically, all the rows with $X = 1$ have $Y = 2$, all the rows with $X = 2$ have $Y = 4$, and all the rows with $X = 3$ have $Y = 4$. In this case, it is possible that X determines Y (aka $X \rightarrow Y$):

X	**Y**
1	2
1	2
1	2
2	4
3	4

Note that in the above example, even though $Y = 4$ for two rows, their X values do not have to be the same to say that X determines Y. We only care if the Y values for a particular X value are the same, not vice versa!

Primary keys are special cases of FDs. A **superkey**[1] is a set of columns that determine all the columns in the table. If X is a superkey of R, then $X \rightarrow R$.

A **candidate key** is a set of columns that determine all the columns in the table such that if we remove any of the columns in a candidate key, the resulting set will not be a superkey of the relation. If X is a candidate key of R, then $X \rightarrow R$ and for every strict subset $Y \subset X$, $Y \nrightarrow R$.

[1] Sometimes we refer to a superkey as a key.

For example, if columns K, L are the columns in a table and K is a primary key of the table (aka column K alone determines all the columns in the table) then KL is superkey and K is a superkey and a candidate key.

We will decompose our relation to avoid redundancies by using functional dependencies. But first, we define the closure of F, denoted as $F+$, to be the set of all FDs that are implied by F. In broader terms, a closure is the full set of table relationships that can be determined from known FDs.

Let's take a look at a couple of examples to better understand these definitions. Given a relation R with columns $\{A, B, C, D, E\}$ and functional dependencies $F = \{B \to CD, D \to E, B \to A, E \to C, AD \to B\}$, let's examine the following questions:

- Is $B \to E$ in $F+$? Yes! To see why, we will evaluate the closure of B. $B+ = \{B, C, D, E, A\}$, we get this because B determines CD from the first functional dependency from the set F. Then we see that D determines E so E is in the closure of B. Thus, from the given implications in F, we see that $B \to E$, so the answer is YES.
- Is D a superkey for relation R? We will determine the answer to this by evaluating the closure of D. $D+ = \{D, E, C\}$. We get this because D determines itself. From the second FD in F, we see that D determines E. Then we see that E determines C. Since $D+$ is not equal to all the columns of the relation R, we know that D is NOT a key for R because knowing D does not determine all columns in R. The answer is NO.
- Is AD a superkey for R? We again evaluate the closure of $AD+$. Using the same procedure as the last two, we get that $AD+ = \{A, D, E, C, B\}$. Therefore, the answer is YES.
- Is AD a candidate key for R? We must see if AD is the minimum subset to imply R by checking the closure of each subset. So we check $A+ = \{A\}$ and $D+ = \{D, E, C\}$. Since none of them are R and that they together make up all columns of R, we know that AD is a candidate key! The answer is YES.
- Is ADE a candidate key for R? We know that AD determines all the columns for R already so ADE is NOT a candidate key. The answer is NO.

3.1 BCNF Decomposition

We will now decompose the relation into Boyce-Codd Normal Form (BCNF). Relation R with FDs F is in BCNF if for all $X \rightarrow A$ in $F+$, either of the following conditions holds true:

- $A \subseteq X$ (called a trivial FD)
- X is a superkey for R (implies $X+ = R$)

The BCNF Decomposition algorithm is as follows:

1. **Input R:** Relation
2. **Input F:** FDs for R
3. $R = \{R\}$
4. If there is a relation $r \in R$ that is not BCNF and has > 2 attributes (note that two attribute relations are always in BCNF):
   (a) Pick a violating FD f: $X \rightarrow A$ such that $X, A \in$ attributes of r.
   (b) Compute $X+$.
   (c) Let $R_1 = X+$. Let $R_2 = X \ \cup \ (r - X+)$.
   (d) Remove r from R.
   (e) Insert R_1 and R_2 into R.
   (f) Recompute F as FDs over all relations $r \in R$.
5. Repeat step 4 until necessary.

Let's take a look at an example to see how this works. Let us look at relation $R = \{C, S, J, D, P, Q, V\}$ with the superkey C and $F = \{JP \rightarrow C, SD \rightarrow P, J \rightarrow S\}$.

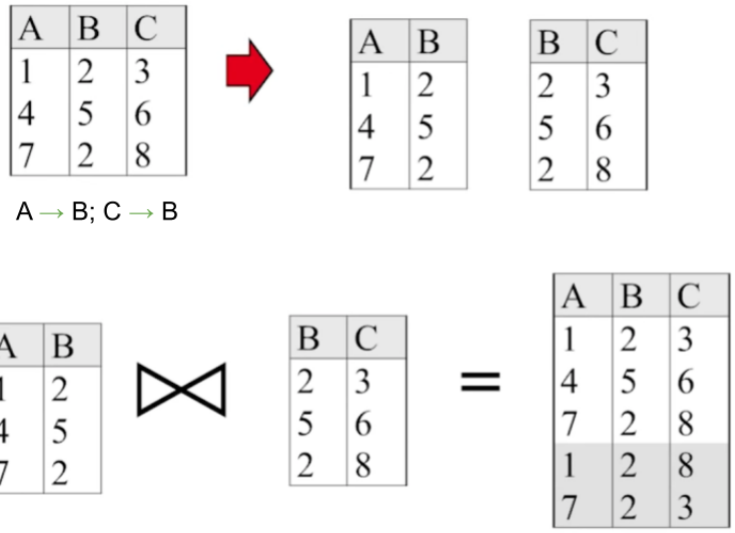
- $JP \rightarrow C$ does not violate the BCNF constraints, since C is a superkey, therefore JP is also a superkey.
- The FD $SD \rightarrow P$ is in violation of BCNF since SD is not a superkey nor is $P \subseteq SD$.
  - Compute $SD+ = \{S, D, P\}$.
  - Decompose R into $R_1 = SD+ = \{S, D, P\}$ and $R_2 = SD \ \cup \ (R - SD+) = \{S, D\} \ \cup \ \{C, J, Q, V\} = \{S, D, C, J, Q, V\}$.
  - We now have 2 relations: $R_1 = \{S, D, P\}$ and $R_2 = \{S, D, C, J, Q, V\}$.
- The FD $J \rightarrow S$ is in violation of BCNF since J is not a superkey for R_2 nor is $S \subseteq J$. R_1 does not have this dependency so it is not decomposed further. Applying the same steps as above we see that R_2 gets decomposed into $\{J, S\}$ and $\{D, C, J, Q, V\}$.

The BCNF form of this relation is SDP, JS, and $DCJQV$.

3.2 Lossless Decomposition

One problem that can arise from decomposing relations is that after breaking up the table into smaller relations, we may be unable to reconstruct the original relation through these decomposed relations. This means that our decomposition is lossy.

For example, consider the following relation R=ABC, which is decomposed into X=AB and Y=BC. This is a lossy decomposition since attempting to reconstruct R by joining X and Y doesn't give us the original relation, but rather a superset of R with some extra junk data. As we can see in this example, lossy decompositions don't "lose" data, but rather generate bad data.



A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B
1	2
4	5
7	2

⋈

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Instead, we want our decompositions to be **lossless**, meaning that if R is decomposed into X and Y, $X \bowtie Y = R$. A decomposition of a relation R into X and Y is lossless with respect to F (a set of FDs) if and only if the closure of F (F^+) contains:

- $X \cap Y \rightarrow X$ ($X \cap Y$ is a superkey of X), or
- $X \cap Y \rightarrow Y$ ($X \cap Y$ is a superkey of Y)

Looking back at our previous example (R = ABC, X=AB, Y=BC), we can apply this test to confirm it is lossy. $X \cap Y = B$, and B is not a superkey of either X or Y (because in X, B = 2 maps to two different values for A; in Y, B = 2 maps to two different values for C).

BCNF decompositions are always lossless, so we don't have to worry about lossiness if we are decomposing a relation into BCNF.

3.3 Dependency Preserving Decompositions

Another problem that can arise from decomposing relations is that dependency checking may require joins. If we make an insertion to a table and need to check that functional dependencies are still satisfied, we may have to join together the tables we decomposed, making every insertion we do costly.

We can avoid this problem through dependency preserving decompositions. Suppose we decompose R into X, Y, and Z. This decomposition is dependency preserving if enforcing FDs individually on each of X, Y, and Z implies that all FDs that held on R must also hold. Now, when making an insertion to a table, we will only need to check the FDs are satisfied on that relation (avoiding joins between decomposed relations) since this will be sufficient for enforcing all FDs that held on the original relation.

Formally, a decomposition of a relation R into X and Y is **dependency preserving** if $(F_X \cup F_Y)^+ = F^+$. If we take the dependencies that can be checked in X without considering Y and the dependencies that can be checked in Y without considering X, the closure of their union will be the closure of the original set of FDs (F^+).

Note that although decompositions into BCNF are lossless, they are not necessarily dependency preserving! To demonstrate this, let's revisit the BCNF decomposition example we examined earlier, where the relation $R = \{C, S, J, D, P, Q, V\}$ with the superkey C and $F = \{JP \rightarrow C, SD \rightarrow P, J \rightarrow S\}$ was decomposed into X = SDP, Y = JS, and Z = $DCJQV$.

Here, $F_X = \{SD \rightarrow P\}$, $F_Y = \{J \rightarrow S\}$, and $F_Z = \{\}$. Thus, $(F_X \cup F_Y \cup F_Z) = \{SD \rightarrow P, J \rightarrow S\}$, and the closure of this set of FDs does not include $JP \rightarrow C$. Thus, this BCNF decomposition is not dependency preserving.

4 Practice Questions

1. Let's say we have two entities, Students and School, connected by the relationship "Attend s". If a student can attend at most one school, but does not have to attend, what kind of line would we want from Students to Attends?

2. From the previous question, we also know that each school has at least one student. What kind of line would we want from School to Attends?

3. Instead, we decide to keep the Student entity but replace school with a Semester entity and connect them with the relationship "Enrolled". A student can be enrolled for up to 10 semesters. What kind of line would we want from Students to Enrolled?

4. We know that D and AB are both superkeys of the table R. Can both of them also be candidate keys? Are they both guaranteed to be candidate keys?
5. Decompose ABCDEFGH into BCNF in the order of the following FDs: $\{G \rightarrow H, EF \rightarrow B, EA \rightarrow C, FH \rightarrow D\}$
6. Decompose ABCDEF into BCNF in the order of the following FDs: $\{A \rightarrow B, B \rightarrow CD, DE \rightarrow F, D \rightarrow A\}$
7. Is the decomposition in #6 lossless? Is it dependency preserving?

5 Solutions

1. We know a student can attend at most one school, so the line between Students and Attends should be a thin arrow.
2. We know each school has at least one student, so the line between School and Attends should be a thick line.
3. We know a student can be enrolled for 0 or more semesters, so the line between Students and Enrolled should be a thin line.
4. Both D and AB are possible candidate keys. AB is a possible candidate key if removing A or B causes it to no longer be a superkey. Only D is guaranteed to be a candidate key. This is because there is nothing we can remove from the set of columns and possibly have a smaller set that is a superkey because D is only one column. On the other hand, AB is not guaranteed to be a candidate key because removing A or B might result in a smaller set that is still a superkey.
5. At step 1, we get ABCDEFG and GH.

   At step 2, we get ACDEFG and EFB and GH.

   At step 3, we get ADEFG and EAC and EFB and GH.
6. First, we consider $A \rightarrow B$. A+ = ABCD so we get ABCD and AEF.

   Next, we consider $B \rightarrow CD$. Since B is a superkey (B+ = ABCD), BCNF is not violated and we do not need to decompose.

   Next, we consider $DE \rightarrow F$. We do not need to decompose as there are no relations with D, E, and F in them.

   Next, we consider $D \rightarrow A$. Since D is a superkey (D+ = ABCD), BCNF is not violated and we do not need to decompose.
7. In #6, we split R = ABCDEF into X = ABCD and Y = AEF. This decomposition is lossless, as all BCNF decompositions are lossless. We can confirm this by noticing that $X \cap Y = A$, and A is a superkey of X.

   However, this decomposition is not dependency preserving. $F_X = \{A \rightarrow B, B \rightarrow CD, D \rightarrow A\}$ and $F_Y = \{\}$. $(F_X \cup F_Y)^+$ does not contain $DE \rightarrow F$, an FD that originally held on R.

CS 186
Spring 2021

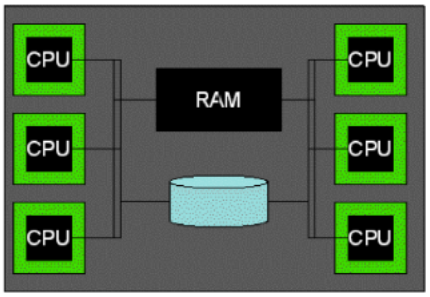
Parallel Query Processing

1 Introduction

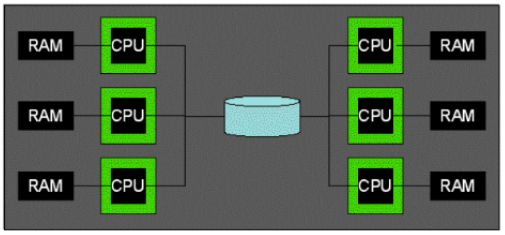
Up until this point we have assumed that our database is running on a single computer. For modern applications that deal with millions of requests over terabytes of data it would be impossible for one computer to quickly respond to all of those requests. We need to figure out how to run our database on multiple computers. We will call this **parallel query processing** because a query will be run on multiple machines in parallel.

2 Parallel Architectures

How are these machines connected together? The most straightforward option would probably be to have every CPU share memory and disk. This is called **shared memory**.

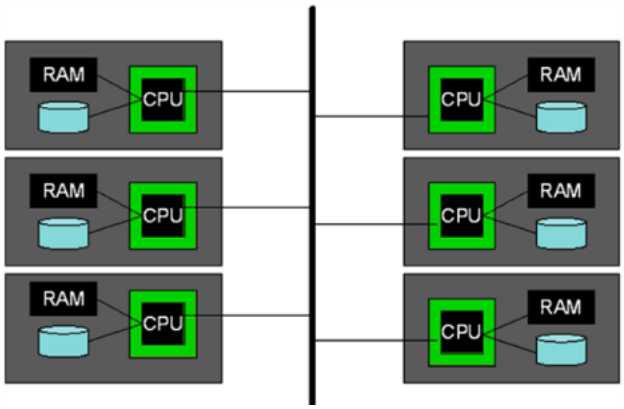


Another option is for each CPU to have its own memory, but all of them share the same disk. This is called **shared disk**.



.

These architectures are easy to reason about, but sharing resources holds back the system. It is possible to achieve a much higher level of parallelism if all the machines have their own disk and memory because they do not need to wait for the resource to become available. This architecture is called **shared nothing** and will be the architecture we use throughout the rest of the note.



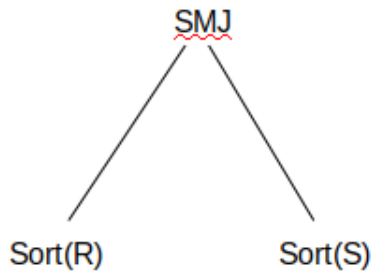
In shared nothing, the machines communicate with each other solely through the network by passing messages to each other.

3 Types of Parallelism

The focus of this note will be on **intra-query** parallelism. Intra-query parallelism attempts to make one query run as fast as possible by spreading the work over multiple computers. The other major type of parallelism is **inter-query** parallelism which gives each machine different queries to work on so that the system can achieve a high throughput and complete as many queries as possible. This may sound simple, but doing it correctly is actually quite difficult and we will address how to do it when we get to the module on concurrency.

3.1 Types of Intra-query Parallelism

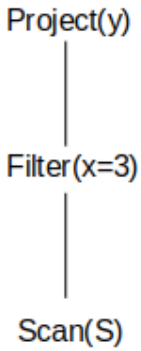
We can further divide Intra-query parallelism into two classes: **intra-operator** and **inter-operator**. **Intra-operator** is making one operator run as quickly as possible. An example of intra-operator parallelism is dividing up the data onto several machines and having them sort the data in parallel. This parallelism makes sorting (one operation) as fast as possible. **Inter-operator** parallelism is making a query run as fast as possible by running the operators in parallel. For example, imagine our query plan looks like this:



One machine can work on sorting R and another machine can sort S at the same time. In inter-operator parallelism, we parallelize the entire query, not individual operators.

3.2 Types of Inter-operator Parallelism

The last distinction we will make is between the different forms of inter-operator parallelism. The first type is **pipeline parallelism**. In pipeline parallelism records are passed to the parent operator as soon as they are done. The parent operator can work on a record that its child has already processed while the child operator is working on a different record. As an example, consider the query plan:



In pipeline parallelism the project and filter can run at the same time because as soon as filter finishes a record, project can operate on it while filter picks up a new record to operate on. The other type of inter-operator parallelism is **bushy tree parallelism** in which different branches of the tree are run in parallel. The example in section 2.1 where we sort the two files independently is an example of bushy tree parallelism.

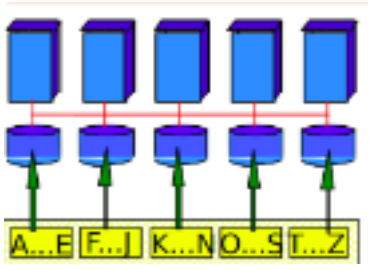
4 Partitioning

Because all of our machines have their own disk we have to decide what data is stored on what machine. In this class each data page will be stored on only one machine. In industry we call this **sharding** and it is used to achieve better performance. If each data page appeared on multiple machines it would be called **replication**. This technique is used to achieve better availability (if one machine goes down another can handle requests), but it presents a host of other challenges that we will not study in depth in this class.

To decide what machines get what data we will use a **partitioning scheme**. A partitioning scheme is a rule that determines what machine a certain record will end up on. The three we will study are range partitioning, hash partitioning, and round-robin.

4.1 Range Partitioning

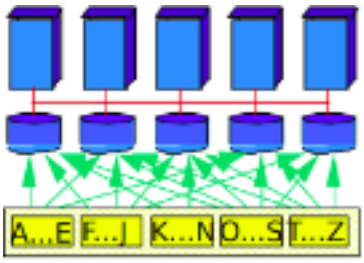
In a **range partitioning** scheme each machine gets a certain range of values that it will store (i.e. machine 1 will store values 1-5, machine 2 will store values 6-10, and so on).



This scheme is very good for queries that lookup on a specific key (especially range queries compared to the other schemes we'll talk about) because you only need to request data from the machines that the values reside on. It's one of the schemes we use for parallel sorting and parallel sort merge join.

4.2 Hash Partitioning

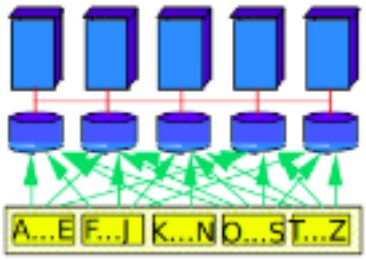
In a **hash partitioning** scheme, each record is hashed and is sent to a machine matches that hash value. This means that all like values will be assigned to the same machine (i.e. if value 4 goes to machine 1 then all of the 4s must go to that machine), but it makes no guarantees about where close values go.



It will still perform well for key lookup, but not for range queries (because every machine will likely need to be queried). Hash partitioning is the other scheme of choice for parallel hashing and parallel hash join.

4.3 Round Robin Partitioning

The last scheme we will talk about is called **round robin partitioning**. In this scheme we go record by record and assign each record to the next machine. For example the first record will be assigned to the first machine, the second record will be assigned to the second machine and so on. When we reach the final machine we will assign the next record to the first machine.



This may sound dumb but it has a nice property - every machine is guaranteed to get the same amount of data. This scheme will actually achieve maximum parallelization. The downside, of course, is that every machine will need to be activated for every query.

5 Network Cost

So far in this class we have only measured performance in terms of IOs. When we have multiple machines communicating over a network, however, we also need to consider the **network cost**. The network cost is how much data we need to send over the network to do an operation. In this class it is usually measured in KB. One important thing to note is that there is no requirement that entire pages must be sent over the network (unlike when going from memory to disk), so it is possible for the network cost to just be 1 record worth of data.

6 Partitioning Practice Questions

Assume that we have 5 machines and a 1000 page `students(sid, name, gpa)` table. Initially, all of the pages start on one machine. Assume pages are 1KB.

1) How much network cost does it take to round-robin partition the table?

2) How many IOs will it take to execute the following query:

```
SELECT * FROM students where name = 'Josh Hug';
```

3) Suppose that instead of round robin partitioning the table, we hash partitioned it on the `name` column instead, How many IOs would the query from part 2 take?

4) Assume that an IO takes 1ms and the network cost is negligible. How long will the query in part 2 take if the data is round-robin partitioned and if the data is hash partitioned on the `name` column?

Now assume in the general case that we have n machines and p pages, all pages start on one machine, and that each pages is k KB. Express answers in terms of n, p, and k.

5) How much network cost does it take to round-robin partition the table?

6) What is the **minimum possible** network cost to hash partition the table?

7) What is the **maximum possible** network cost to hash partition the table?

8) Now assume pages are randomly distributed across machines instead of all starting on one machine. Let the ith machine contain p_i pages such that $\sum_{i=1}^{n} p_i = p$. How much network cost does it take to hash partition the table across all machines in the **average** case?

7 Partitioning Practice Solutions

1) In round robin partitioning the data is distributed completely evenly. This means that the machine the data starts on will be assigned 1/5 of the pages. This means that 4/5 of the the pages will need to move to different machines, so the answer is: 4/5 * 1000 * 1 = **800 KB**.

2) When the data is round robin partitioned we have no idea what machine(s) the records we need will be on. This means we will have to do full scans on all of the machines so we will have to do a total of **1000 IOs**.

3) When the data is hash partitioned on the name column, we know exactly what machine to go to for this query (we can calculate the hash value of 'Josh Hug' and find out what machine is assigned that hash value). This means we will only have to a full scan over 1 machine for a total of **200 IOs**. *Of course, this assumes all records with value 'Josh Hug' can fit on one machine.*

4) Under both partitioning schemes it will take **200ms**. Each machine will take the same amount of time to do a full scan, and all machines can be run at the same time so for runtime it doesn't matter how many machines we need to query[1].

5) We send $\frac{n-1}{n}$ of the pages to other machines, leaving $\frac{1}{n}$ at the starting machine. Therefore, the network cost is then $\frac{pk(n-1)}{n}$.

6) In the minimum possible network cost, all tuples hash to the starting machine so we do not need to send any tuples to other machines, resulting in a minimum network cost of **0**.

7) In the minimum possible network cost, all tuples hash to other machines machine so we need to send all tuples to other machines, resulting in a maximum network cost of **pk**.

[1] In reality, there may be some variance in how long each machine takes to complete a scan, and the likelihood of a "straggler" (i.e. slow machine) increases with the number of machines involved in a query, but for simplicity we ignore this detail

8) In the average case, each machine will hash $\frac{n-1}{n}$ of its pages to other machines, resulting in a network cost of $\frac{n-1}{n}\sum_{i=1}^{n} p_i k = \frac{pk(n-1)}{n}$.

8 Parallel Sorting

Let's start speeding up some algorithms we already know by parallelizing them. There are two steps for parallel sorting:

1. Range partition the table
2. Perform local sort on each machine

We range partition the table because then once each machine sorts its data, the entire table is in sorted order (the data on each machine can be simply concatenated together if needed).

9 Parallel Hashing

Parallel hashing is very similar to parallel sorting. The two steps are:

1. Hash partition the table
2. Perform local hashing on each machine

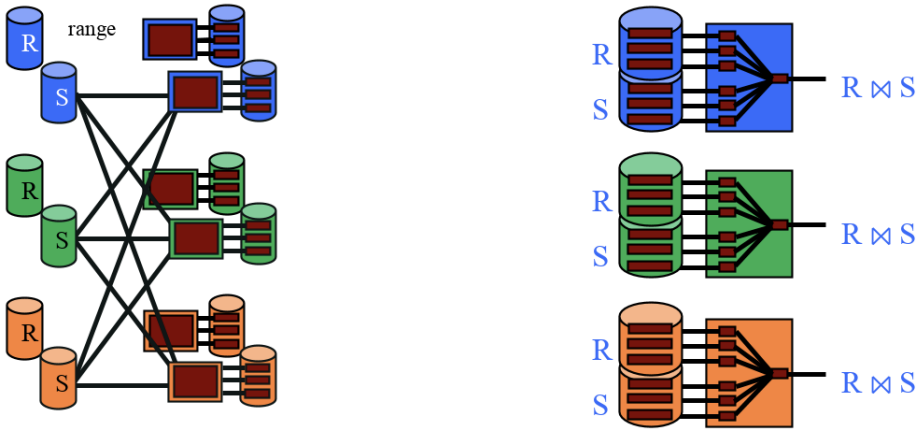
Hash partitioning the table guarantees that like values will be assigned to the same machine. It would also be valid to range partition the data (because it has the same guarantee). However, in practice, range partitioning is a little harder (how do you come up with the ranges efficiently?) than hash-partitioning so use hash partitioning when possible.

10 Parallel Sort Merge Join

The two steps for parallel sort merge join are:

1. Range partition each table **using the same ranges** on the join column
2. Perform local sort merge join on each machine

We need to use the same ranges to guarantee that all matches appear on the same machine. If we used different ranges for each table, then it's possible that a record from table R will appear on a different machine than a record from table S even if they have the same value for the join column which will prevent these records from ever getting joined.

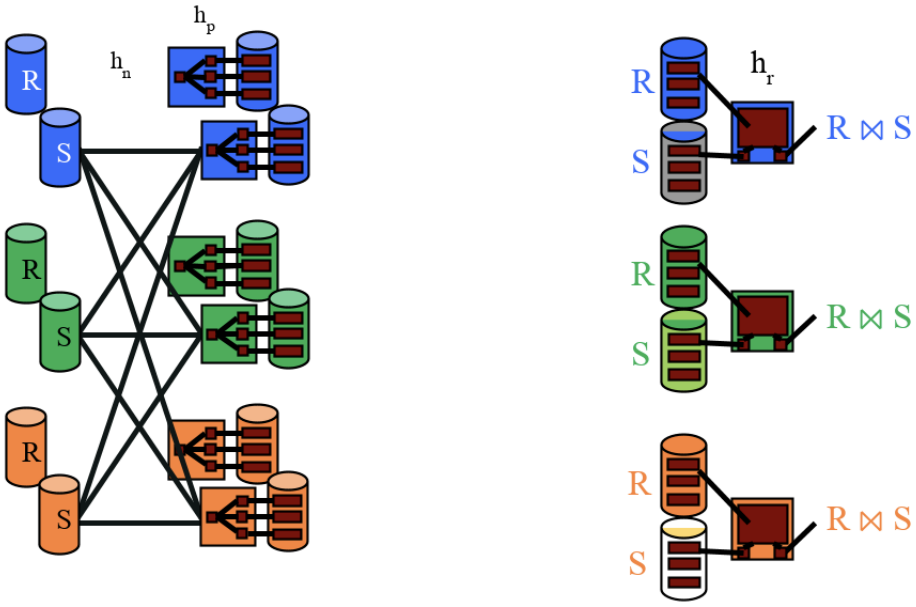


11 Parallel Grace Hash Join

The two steps for parallel hash join are:

1. Hash partition each table **using the same hash function** on the join column
2. Perform local grace hash join on each machine

Similarly to parallel SMJ, we need to use the same hash function for both tables to guarantee that matching records will be partitioned to the same machines.



12 Broadcast Join

Let's say we want to join a 1,000,000 page table that is currently round robin partitioned with a 100 page table that is currently all stored on one machine. We could do one of the parallel join algorithms discussed above, but to do this we would need to either range partition or hash partition the tables. It will take a ton of network cost to partition the big table.

Instead, we can use a **broadcast join**. A broadcast join will send the entire small relation to every machine, and then each machine will perform a local join. The concatenation of the results from each machine will be the final result of the join. While this algorithm has each machine operate on more data, we make up for this by sending much less data over the network. When we're joining together one really large table and one small table, a broadcast join will normally be the fastest because of its network cost advantages.

13 Symmetric Hash Join

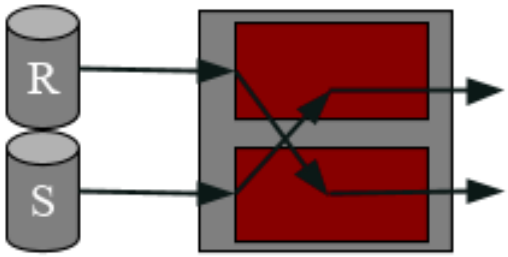
The parallel joins we have discussed so far have been about making the join itself run as fast as possible by distributing the data across multiple machines. The problem with the join algorithms we have discussed so far is that they are **pipeline breakers**. This means that the join cannot produce output until it has processed every record. Why is this a problem? It prevents us from using pipeline parallelism. The operators above cannot work in parallel to the join because the join is taking a lot of time and then producing all of the output at once.

Sort Merge Join is a pipeline breaker because sorting is a pipeline breaker. You cannot produce the first row of output in sorting until you have seen all the input (how else would you know if your row is really the smallest?). Hash Join is a pipeline breaker because you cannot probe the hash table of each partition until you know that hash table has all the data in it. If you tried to probe one of the hash tables after processing only half the data, you might get only half of the matches that you should! Let's try to build a hash based join that isn't a pipeline breaker.

Symmetric Hash Join is a join algorithm that is pipeline-friendly, we can start producing output as soon as we see our first matches. Here are the steps to symmetric hash join:

1. Build two hash tables, one for each table in the join
2. When a record from R arrives, probe the hash table for S for all of the matches. When a record from S arrives, probe the hash table for R for all of the matches.
3. Whenever a record arrives add it to its corresponding hash table after probing the other hash table for matches.

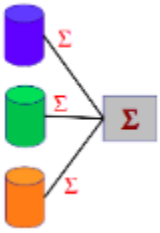
This works because every output tuple gets generated exactly once - when the second record in the match arrives! But we produce output as soon as we see a match so it's not a pipeline breaker.



14 Hierarchical Aggregation

The final parallel algorithm we'll discuss is called **hierarchical aggregation**. Hierarchical aggregation is how we parallelize aggregation operations (i.e. SUM, COUNT, AVG). Each aggregation is implemented differently, so we'll only discuss two in this section, but the ideas should carry over to the other operations easily.

To parallelize COUNT, each machine individually counts their records. The machines all send their counts to the coordinator machine who will then sum them together to figure out the overall count.



AVG is a little harder to calculate because the average of a bunch of averages isn't necessarily the average of the data set. To parallelize AVG each machine must calculate the sum of all the values and the count. They then send these values to the coordinator machine. The coordinator machine adds up the sums to calculate the overall sum and then adds up the counts to calculate the overall count. It then divides the sum by the count to calculate the final average.

15 Parallel Algorithms Practice Questions

For 1 and 2, assume we have a 100 page table R that we will join with a 400 page table S. All the pages start on machine 1, and there are 4 machines in total with 30 buffer pages each.

1) How many passes are needed to do a parallel unoptimized SMJ on the data? For this question a pass is defined as a full pass over either table (so if we have to do 1 pass over R and 1 pass over S it is 2 total passes).

2) How many passes are needed to a parallel Grace Hash Join on the data?

3) We want to calculate the max in parallel using hierarchical aggregation. What value should each machine calculate and what should the coordinator do with those values?

4) If instead we had a 10000 page table R round-robin partitioned across 10 machines and the same 400 page table S on machine 1, what type of join would be appropriate for an equijoin?

5) If we used a non-uniform hash function which resulted in an uneven hash partition of our records, how will this affect the total time it takes to run parallel Grace Hash Join?

6) Consider a parallel version of Block Nested Loop Join in which we first range partition tuples from both tables and then perform BNLJ on each individual machine. Is this parallel algorithm pipeline friendly?

16 Parallel Algorithms Practice Solutions

1) 2 passes to partition the data (1 for each table). Each machine will then have 25 pages for R and 100 pages for S. R can be sorted in 1 pass but S requires 2 passes. Then it will take 2 passes to merge the tables together (1 for each table). This gives us a total of 2 (partition) + 1 (sort R) + 2 (sort S) + 2 (merge relations) = **7 passes**.

2) Again we will need 2 passes to partition the data across the four machines and each machine will have 25 pages for R and 100 for S. We don't need any partitioning passes because R fits in memory on every machine, so we only need to do build and probe, which will take 2 passes (1 for each table). This gives us a total of **4 passes**.

3) Each machine should calculate the max and the coordinator will take the max of those maxes.

4) A broadcast join since this type of join helps reduce network costs when we have two tables with drastically different sizes and the larger one is split into many machines.

5) The parallel join algorithm would only finish when the last machine finishes its Grace Hash Join. This last machine is likely to be the machine that received the most pages from the partitioning phase since it may take more I/Os to hash it.

6) No, because if BNLJ runs for each block of the outer relation R, we need scan through the

inner relation S to find matching tuples and thus, we would need the machine to have all of S first before we can begin the join. Therefore, this is a pipeline breaker.

CS 186
Spring 2021

Distributed Transactions

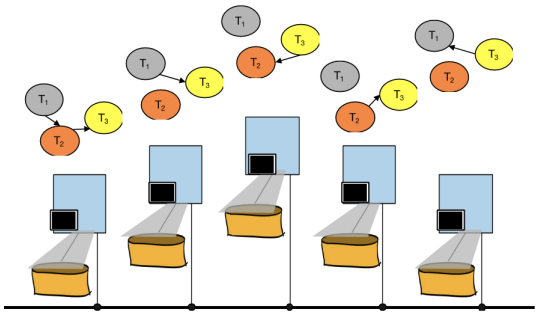
1 Introduction

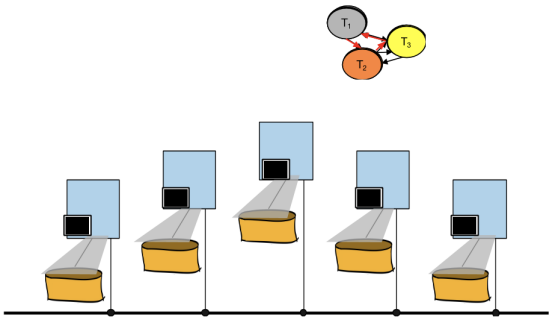
For much of this semester, we assumed that transactions ran on databases where all of the data existed on one node (machine). This is often the case for databases with lighter workloads, but as demand increases, databases scale out to improve performance by using a **Shared Nothing architecture**. Each node receives a partition of the data set that is distributed based on a range or hash key and is connected to other nodes through a network. Distributed Transactions are needed for executing queries in distributed databases as a transaction may need to perform reads and writes on data that exist on different nodes.

2 Distributed Locking

Since every node contains data that is independent of any other node's data, every node can **maintain its own local lock table**. Coarser grained locks for entire tables or the database can either be given to all nodes containing a partition or be centralized at a predetermined node. This design makes locking simple as **2 phase locking** is performed at every node using local locks in order to guarantee serializability between different transactions.

When dealing with locking, deadlock is always a possibility. To determine whether deadlock has occurred in a distributed database, the waits-for graphs for each node must be unioned to find cycles as transactions can be blocked by other transactions executing on different nodes.





3 Two Phase Commit (2 PC)

In a distributed database, **consensus is the idea that all nodes agree on one course of action**. Consensus is implemented through Two Phase Commit and enforces the property that all

nodes maintain the same view of the data. It provides this guarantee by ensuring that a distributed transaction either commits or aborts on all nodes involved. If consensus is not enforced, some nodes may commit the transaction while others abort, causing nodes to have views of data at different points in time.

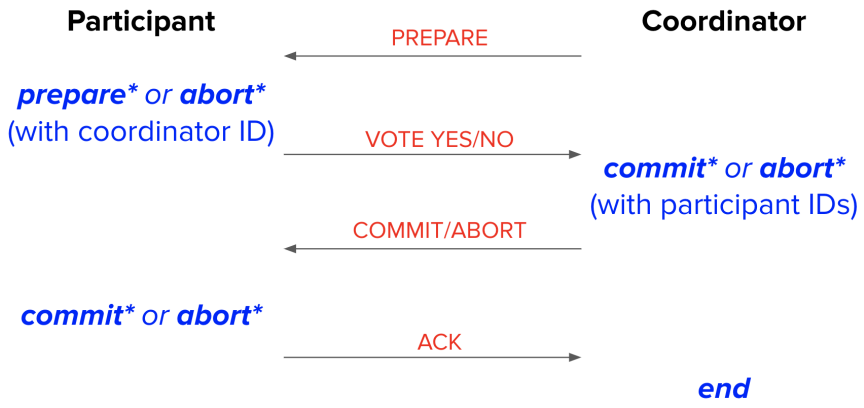
Every distributed transaction is assigned a coordinator node that is responsible for maintaining consensus among all participant nodes involved in the transaction. When the transaction is ready to commit, the coordinator initiates Two Phase Commit.

2 PC's first phase is the **preparation phase**:

1. Coordinator sends prepare message to participants to tell participants to either prepare for commit or abort
2. Participants generate a prepare or abort record and flush record to disk
3. Participants send yes vote to coordinator if prepare record is flushed or no vote if abort record is flushed
4. Coordinator **generates a commit record if it receives unanimous yes votes** or an abort record otherwise, and flushes the record to disk

2 PC's second phase is the **commit/abort phase**:

1. Coordinator broadcasts (sends message to every participant) the result of the commit/abort vote based on flushed record
2. Participants generate a commit or abort record based on the received vote message and flush record to disk
3. Participants send an ACK (acknowledgement) message to the coordinator
4. Coordinator generates an end record once all ACKs are received and flushes the record sometime in the future



4 Distributed Recovery

It is also important that the Two-Phase Commit protocol maintains consensus among all nodes ***even in the presence of node failures***. In other words – suppose a node were to fail at an arbitrary point in the protocol. When this node comes back online, it should still end up making the same decision as all the other nodes in the database.

How do we accomplish this? Luckily, the 2PC protocol tells us to log the prepare, commit, and abort records. This means that between looking at our own log, and talking to the coordinator node, we will have all the information needed to accomplish recovery correctly (for most failure scenarios).

An important assumption we will make for now is that failures are temporary and that all nodes eventually recover. Many of 2PC's invariants break down without this assumption, but that is beyond the scope of this class.

Let's look at the specifics. We will analyze what happens for a failure at each possible point in the protocol, for either the participant or the coordinator.

Note that in some cases, we can determine what recovery decisions to make just by looking at our own log. In other cases, however, we might also have to talk to the coordinator; we wrap this logic in a separate process called the *recovery process*.

The possible failures, in chronological order:

- **Participant** is recovering, and sees **no prepare record**.

  - This probably means that the participant has not even started 2PC yet – and if it has, it hasn't yet sent out any *vote messages* (since votes are sent after flushing the log record to disk).
  - Since it has not sent out any vote messages, it **aborts** the transaction locally. **No messages need to be sent out** (the participant has no knowledge of the coordinator ID).

- **Participant** is recovering, and sees a **prepare record**.
  - This situation is trickier. Looking at the diagram above, a lot of things could have happened between logging the prepare record and crashing – for instance, we don't even know if we managed to send out our YES vote!
  - Specifically, we don't know whether or not the coordinator made a commit decision. So the participant node's recovery process must ask the coordinator whether a commit happened ("Did the coordinator log a commit?"). The coordinator can be determined from the coordinator ID stored in the prepare log record.
  - The coordinator will respond with the commit/abort decision, and the **participant resumes 2PC from phase 2**.

- **Coordinator** is recovering, and sees **no commit record**.
  - The coordinator crashed at some point before receiving the votes of all participants and logging a commit decision.
  - The coordinator will **abort** the transaction locally. **No messages need to be sent out** (the coordinator has no knowledge of the participant IDs involved in the transaction).
  - If the coordinator receives an inquiry from a participant about the status of the transaction, respond that the transaction aborted.

- **Coordinator** is recovering, and sees a **commit record**.
  - We'd like to commit, but we don't know if we managed to tell the participants.
  - So, **rerun phase 2** (send out commit messages to participants). The participants can be determined from the participant IDs stored in the commit log record.

- **Participant** is recovering, and sees a **commit record**.
  - We did all our work for this commit, but the coordinator might still be waiting for our ACK, so **send ACK to coordinator**. (The coordinator can be determined from the coordinator ID stored in the commit log record.)

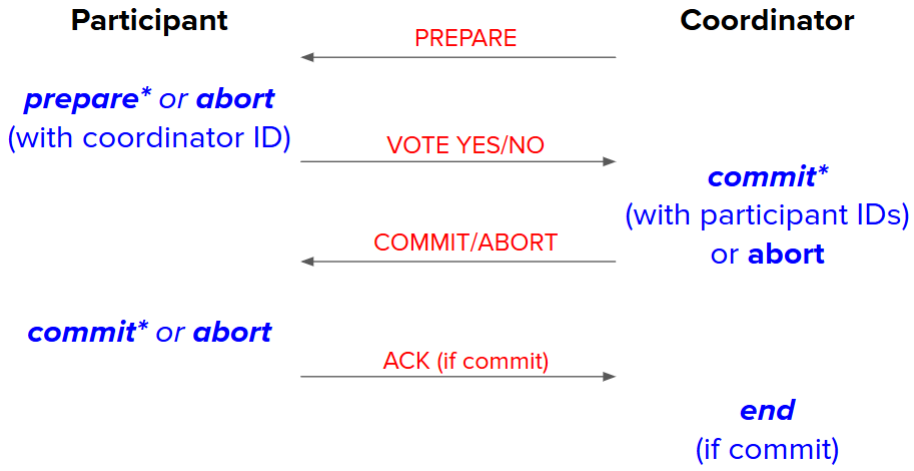
- **Coordinator** is recovering, and sees an **end record**.
  - This means that everybody already finished the transaction and there is no recovery to do.

The list above only discusses commit records. **What about abort records?**

We could handle them the same way as commit records (e.g. tell participants, send acks). This works fine, but is it really necessary? What if nodes just didn't bother recovering aborted transactions?

It turns out this works if everybody understands that **no log records means abort**. This optimization is called **presumed abort**, and it means that **abort records never have to be flushed** – not in phase 1 or phase 2, not by the participant or the coordinator.

The protocol with the presumed abort optimization looks like this:



And this is how we would recover from failures in aborted transactions, with and without presumed abort:

- **Participant** is recovering, and sees **no phase 1 abort record**.
  - Without presumed abort: This probably means that the participant has not even started 2PC yet – and if it has, it hasn't yet sent out any *vote messages* (since votes are sent after flushing the log record to disk).
  - With presumed abort: It is possible that the participant decided to abort and sent a "no" vote to the coordinator before the crash.
  - With or without presumed abort, the participant **aborts** the transaction locally. **No messages need to be sent out** (the participant has no knowledge of the coordinator ID).
- **Participant** is recovering, and sees a **phase 1 abort record**.

  - Without presumed abort: Abort the transaction locally and send "no" vote to the coordinator. (The coordinator can be determined from the coordinator ID stored in the abort log record.)
  - With presumed abort: Abort the transaction locally. No messages need to be sent out! (The coordinator will timeout after not hearing from the participant and presume abort.)

- **Coordinator** is recovering, and sees **no abort record**.
  - Without presumed abort: The coordinator crashed at some point before reaching a commit/abort decision.
  - With presumed abort: It is possible that the coordinator decided to abort and sent out abort messages to the participants before the crash.
  - With or without presumed abort, the coordinator will **abort** the transaction locally. No messages need to be sent out (the coordinator has no knowledge of the participant IDs involved in the transaction).
  - If the coordinator receives an inquiry from a participant about the status of the transaction, respond that the transaction aborted.

- **Coordinator** is recovering, and sees an **abort record**.
  - Without presumed abort: Rerun phase 2 (sending out abort messages to participants). The participants can be determined from the participant IDs in the abort log record.
  - With presumed abort: Abort the transaction locally. No messages need to be sent out! (Participants who don't know the decision will ask the coordinator later.)

- **Participant** is recovering, and sees a **phase 2 abort record.**[1]
  - Without presumed abort: Abort the transaction locally, and send back ACK to coordinator. (The coordinator can be determined from the coordinator ID stored in the abort log record.)
  - With presumed abort: Abort the transaction locally. No messages need to be sent out! (ACKs only need to be sent back on commit.)

To wrap everything up, here are some subtleties that you should be explicitly be aware of:

- The 2PC recovery decision is **commit *if and only if* the coordinator has logged a commit record**.

[1] In reality, there is **no difference** between a phase 1 and phase 2 abort record. Therefore, for the without presumed abort case, a participant can simply respond with NO to the coordinator upon seeing an abort record. The coordinator will then treat that message as either a phase 1 vote or a phase 2 ACK accordingly.

- Since 2PC requires unanimous agreement, it will only make progress if all nodes are alive. This is true for the recovery protocol as well – for recovery to finish, all failed nodes must eventually come back alive. If the coordinator believes a participant is dead, it can respawn the participant on a new node based on the log of the original participant, and ignore the original participant if it does come back online. However, 2PC struggles to handle scenarios where the coordinator is dead. For example, consider a scenario where all participants vote yes in Phase 1, but the coordinator crashes before sending out a commit decision. The participants will keep pinging the dead coordinator for the status of the transaction, and the system is blocked from making progress. Protocols that continue operating despite extended failures are out of the scope of this course, but a good example is *"Paxos Commit"*.

5 Practice Questions

1. Suppose that there are some transactions happening concurrently. If no two concurrent transactions ever operate on rows that are being stored in the same node, can a deadlock still happen?

2. Suppose a participant receives a prepare message for a transaction. Assuming the participant node remains online and does not fail, why might it decide to log an abort record and vote no?

3. Suppose a participant receives a prepare message for a transaction and replies `VOTE-YES`. Suppose that they were running a wound-wait deadlock avoidance policy, and a transaction comes in with higher priority. Will the new transaction abort the prepared transaction?

4. True or false - if we are recovering and see a `PREPARE` record, it means we must have sent out a `YES` vote.

5. How many messages and log flushes does the presumed abort optimization skip if the transaction commits? What about if it aborts due to the participants aborting?

6 Solutions

1. You may be inclined to say no, since a transaction does not seem like it will ever wait on the locks of another transaction in this scenario. But remember that in addition to each node maintaining its own local lock table, **coarser grained locks, such as locks for tables or the entire database, can still be shared between nodes**, and the transactions may still conflict via these coarser locks.

2. The participant may decide to abort the transaction and vote no to avoid deadlock. For example, if we are using deadlock detection and we discover that the union of the waits-for graphs for each node contains a cycle the transaction is involved in, we may decide to abort the transaction. If we are using a deadlock avoidance policy, we will abort the transaction if acquiring the locks necessary to perform the transaction's operations would require us to wait on a lower priority transaction (for wait-die) or higher priority transaction (for wound-wait).

3. No - transactions that are prepared **must** be ready to commit if the coordinator tells them to, so they cannot be aborted by anyone other than the coordinator.

4. False! We could have crashed between logging `PREPARE` and sending `VOTE-YES`.

5. If the transaction commits, no messages or log flushes are skipped - the messages sent and records logged are the same as the protocol without presumed abort.

   If the participants abort, they do not flush any log records (skipping up to two flushes) and only have to send one message (they don't have to send the ACK). The coordinator also does not have to flush any log records (skipping one flush), but does not skip any messages (it still sends the messages it would without presumed abort).

CS 186
Spring 2021

NoSQL

1 Introduction

Most of the semester has focused on the properties and implementation of **relational database management systems**, but here we will take a detour to explore a relatively new[1] class of database that has come to be called **NoSQL**. NoSQL databases deviate from the set of database design principles that we have studied in order to achieve the high scale and elasticity needed for modern "Web 2.0" applications. In this document we will explore the motivation and history of NoSQL databases, review the design and properties of several types of NoSQL databases, compare the NoSQL data model with the relational data model, and conclude with a description of MongoDB, a popular and representative NoSQL database.

2 Motivation and History

2.1 OLTP and OLAP

In order to understand the emergence and properties of NoSQL, we must first describe two broad classes of workloads that can be subjected to a database.

Online Transaction Processing (OLTP) is a class of workloads characterized by high numbers of transactions executed by large numbers of users. This kind of workload is common to "frontend" applications such as social networks and online stores. Queries are typically simple lookups (e.g., "find user by ID", "get items in shopping cart") and rarely include joins. OLTP workloads also involve high numbers of updates (e.g., "post tweet", "add item to shopping cart"). Because of the need for consistency in these business-critical workloads, the queries, inserts and updates are performed as transactions.

Online Analytical Processing is a class of *read-only* workloads characterized by queries that typically touch a large amount of data. OLAP queries typically involve large numbers of joins and aggregations in order to support decision making (e.g., "sum revenues by store, region, clerk, product, date").

In many cases, OLTP and OLAP workloads are served by separate databases. Data must be migrated from OLTP systems to OLAP systems every so often via a process called *extract-transform-load* (ETL).

2.2 NoSQL: Scaling Databases for Web 2.0

In the early 2000s, web applications began to incorporate more user-generated content and interactions between users (called "Web 2.0"). Databases needed to handle the very large scale OLTP workload consisting of the tweets, posts, pokes, photos, videos, likes and upvotes being inserted and queried by millions or tens of millions of users simultaneously. Database designers began exploring how to meet this required scale by relaxing the guarantees and reducing the functionality provided

[1] and also old, as we will see.

NoSQL

Property	OLTP	OLAP
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import or event stream
Primarily used by	End user/customer, via web application	Internal analysis, for decision support
What data represents	Latest state of data (current point in time)	History of events that happend over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Table 1: Table from *Designing Data-Intensive Applications* by Martin Kleppmann comparing characteristics of OLTP and OLAP.

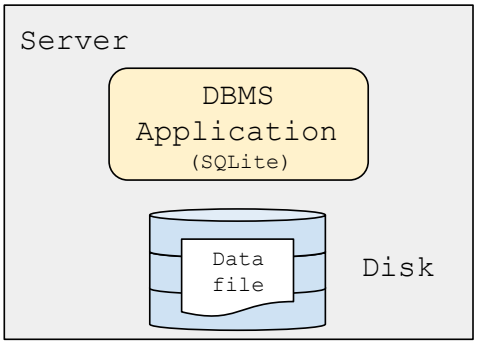


Figure 1: A "single-tier" architecture: one server contains both the database (file) and the application

by relational databases. The resulting databases, termed NoSQL, exhibit a simpler data model with restricted updates but can handle a higher volume of simple updates and queries.

The natural question to ask is: How does simplifying the data model and reducing the functionality of the database allow NoSQL to scale better than traditional relational databases? More specifically, why is it hard to scale relational databases? We can illustrate the answers to these questions through the history of techniques for scaling relational databases.

One-tier architecture (Figure 1): *In the beginning*, before clusters of computers were common place,[2] a database and its application code ran on a shared, single machine. If your application needed to scale — e.g., more storage, faster transaction processing — you simply bought a more powerful machine. The advantage of this setup is that enforcing *consistency* for OLTP workloads is relatively straightforward. There is a single database (often stored in a single file like SQLite), and because there is only a single application using the database at a time, there was little need for concurrency control to maintain consistency, which drastically simplified the implementation. However, this architecture is only appropriate for a few scenarios, namely those with a single

[2]Seymour Cray famously asked ""If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"

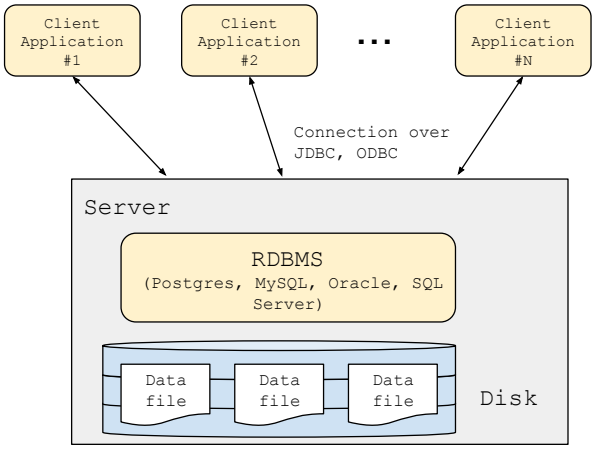


Figure 2: A "two-tier" architecture: the database resides on a single server and multiple client applications connect to it

application that needs to interact with the database.

Two-tier architecture (Figure 2): The need to support multiple simultaneous applications led to the emergence of *client-server architecture*. In this setup, there is a single database server (typically on a powerful server or cloud service) that is accessed by multiple applications. Each application (referred to as a *client*) connects to the database using an application programming interface like JDBC or ODBC. Because the actions of these multiple applications may conflict with one another, transactions are imperative to maintain consistency of the database. This architecture permits a larger number of applications to access the same database, and enables higher transaction throughput (the number of transactions the system is able to serve per unit time). However, there is still just one database server and one application server (per application).

Three-tier architecture (Figure 3): As one might imagine, one application server is not sufficient to meet the load of modern, global-scale web applications like modern social networks and online stores. In the two-tier architecture the application server is considered the client, so there are maybe 10s of clients connecting to the database. For modern web applications the user's browser is considered the client, meaning there are 10s to 100s of millions of clients that need to be served. Application servers can be replicated in order to meet this scale. Application servers are relatively easy to replicate because they do not share state (i.e., the only state they need to keep is which clients are connected to them), so it is possible to add 100s or 1000s of new application nodes in order to meet demand. All common state is handled by a database "backend" which communicates with the application servers. This places a high OLTP load on the database, which quickly outstrips the capabilities of a single database server.

Now we must examine *how* to scale the database beyond a single server while still providing the *consistency* required by the OLTP workloads.

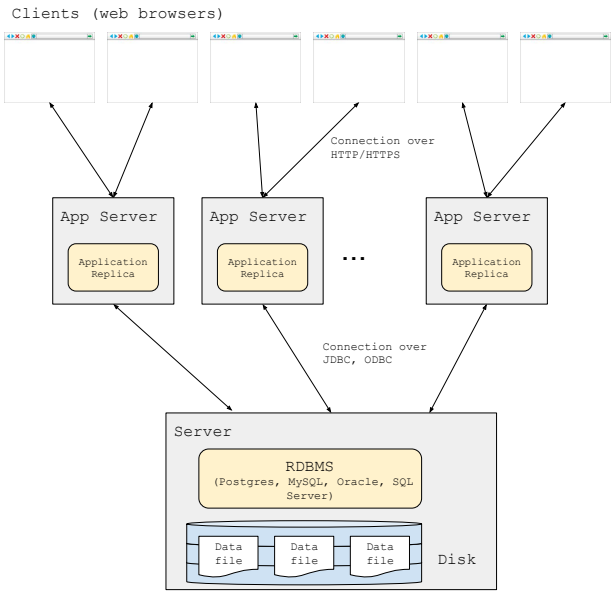


Figure 3: A "three-tier" architecture: database server(s) are distinct from the application frontend servers, which serve user traffic directly. The frontend can be scaled by adding more application servers.

3 Methods for Scaling Databases

3.1 Partitioning

The first mechanism we can leverage for scaling a database is partitioning. We have studied partitioning (also called sharding) before when discussing parallel query processing. In order to meet the performance needs of a modern, web-scale application, partitioning becomes extremely useful, as the database can be partitioned into segments of data that are spread out over multiple machines. Performance can increase for two reasons: 1) queries can be executed in parallel on multiple machines if they touch different parts of the database and 2) partitioning the database may allow each partition to fit into memory, which can reduce the disk I/O cost for executing queries. Partitioning can be effective for write-heavy workloads, as query write operations such as inserts and updates, will likely involve writing to just a single machine. However, read-heavy workloads may suffer when queries access data spanning multiple machines (e.g. consider a join on relations partitioned across many machines, requiring costly network transfer), which can decrease performance in comparison to the single database server model.

3.2 Replication

Replication is motivated not only by the need to scale the database, but also for the database (and by association the application dependent on accessing the database) to be resilient to failures and extensive down-time where the machine is unresponsive. In all of the 3 architectures studied in Section 2, the database server is a *single point of failure*, a component that upon failure stops the entire system from functioning. With replication, the data is replicated on multiple machines. This is effective for read-heavy workloads, as queries that read the same data can be executed in parallel on different replicas. However, as the number of replicas increases, writes become increasingly expensive, as queries that update data must now write to each replica of the data to keep them in sync. For each partition, there is a main/primary (formerly called master) copy and duplicates/replicas that are kept in sync.

Partitioning and replication are often used together in real systems to leverage the performance benefits and to make the system more *fault-tolerant*. As the system scales with more replicas and more partitions, it becomes important to ask questions about how the system will respond when machines begin to fail and the network makes no guarantees about ordering or even the successful delivery of messages.

3.3 CAP Theorem

When designing distributed systems, there are three desirable properties that we define here:

- **Consistency:** The distributed systems version of consistency is a different concept from the consistency found in ACID. The C in ACID refers to maintaining various integrity constraints of a relational database, such as ensuring that a table's primary key constraint is satisfied. On the other hand, consistency in the distributed systems context refers to ensuring that two clients making simultaneous requests to the database should get the same view of the data. In other words, even if each client connects to a different replica, the data that they receive should be the same.
- **Availability:** For a system to be available, every request must receive a response that is not an error, unless the user input is inherently erroneous.
- **Partition Tolerance:** A partition tolerant system must continue to operate despite messages between machines being dropped or delayed by the network, or disconnected from the network altogether.

The CAP theorem (also known as Brewer's Theorem)[3] proves that it is impossible for a distributed system to simultaneously provide more than two of the three (CAP) properties defined

[3]The CAP theorem was first presented as a conjecture by Eric Brewer in 1998 and later formally proved by Seth Gilbert and Nancy Lynch in 2002.

above.[4] In reality, it is almost impossible to design a system that is completely safe from network failures, and thus most systems must be designed with partition tolerance in mind. The tradeoff then comes between choosing consistency or availability during these periods where the network is partitioned. It is important to note that if there is no network failure, it is possible to provide both consistency and availability.

A system that chooses consistency over availability will return an error or time-out if it cannot ensure that the data view it has access to contains the most recent updates. On the other hand, a system that chooses availability over consistency will always respond with the data view it has access to, even if it is unable to ensure that it contains the most recent updates. In a system that chooses availability over consistency, it is possible that two clients that send a simultaneous read request to different replica machines will receive different values in the event of a network failure that prevents messages between those two replicas. As a concrete example contrasting the two system design choices outlined above, consider a client reading from a replica that cannot access the main copy/replica (due to a network failure). The client will either receive an error (prioritizing consistency) or a stale copy (prioritizing availability).

In fact many such systems provide *eventual consistency* instead of consistency, in order to trade off availability and consistency. Eventual consistency guarantees that eventually all replicas will become consistent once updates have propagated throughout the system. Eventual consistency allows for better performance, as write operations no longer have to ensure that the update has been successful on all replicas. Instead, the update can be propagated to the rest of the replicas asynchronously. This does lead to a concern about how simultaneous updates at different replicas are resolved, but this is beyond the scope of this class.

So given the difficulty in scaling relational systems, application designers started looking into alternate data models, and that's where NoSQL comes in.

4 NoSQL Data Models

NoSQL data stores encompass a number of different data models that differ from the relational model.

4.1 Key-Value Stores

The key-value store (KVS) data model is extremely simple and consists only of (key, value) pairs to allow for flexibility. The key is typically a string or integer that uniquely identifies the record and the value can be chosen from a variety of field types. It is typical for a KVS to only allow for byte-array values, which means the application has the responsibility of serializing/deserializing various data types into a byte-array. Due to the flexibility of the KVS data model, a KVS cannot perform operations on the values, and instead provides just two operations: *get(key)* and *put(key, value)*.

[4] For a concise, illustrated proof of the CAP theorem written by a former TA, refer to `https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/`

Examples of popular key-value stores used in industry include AWS's DynamoDB, Facebook's RocksDB, and Memcached.

4.2 Wide-Column Stores

Wide-column stores (or extensible record stores) offer a data model compromise between the structured relational model and the complete lack of structure in a key-value store. Wide-column stores store data in tables, rows, and dynamic columns. Unlike a relational database, wide-column stores do not require each row in a table to have the same columns. Another way to reason about wide-column stores is as a 2-dimensional key-value store, where the first key is a row identifier used to find a particular row, and the second key is used to extract the value of a particular column. The data model has two options:

- key = rowID, value = record
- key = (rowID, columnID), value = field

The only operations provided are the same as a KVS: *get(key)* (or *get(key, [columns])*, which can be thought of as performing a projection on the record) and *put(key, value)*. A couple of the most popular wide-column stores used in industry include Apache Cassandra and Apache HBase (which is based on Google's BigTable).

4.3 Document Stores

In contrast with key-value stores, which are the least-structured NoSQL data model, document stores are among the most structured. The "value" in key-value stores is often a string or byte-array (called *unstructured data*): this is maximally flexible (you can store anything in a byte array), but often it can be helpful to adopt some convention. This is referred to as *semi-structured data*. Key-value stores whose values adhere to a semi-structured data format such as JSON, XML or Protocol Buffers are termed **document stores**; the values are called **documents**. Relational databases store tuples in tables; document databases store documents in collections.

5 Document vs Relational Data Models

Recall that the relational data model organizes length-n tuples into unordered collections called tables, which are described by a set of attributes; each attribute has a name and a datatype. Document data models, on the other hand, can express more complex data structures such as maps/dictionaries, list/arrays and primitive data types, which may be arranged into nested structures. Structured data formats used in document stores include (but are not limited to) Protocol Buffers[5], XML[6] and JSON.[7]

[5] Also called "protobuf": `https://developers.google.com/protocol-buffers`
[6] Extensible Markup Language: `https://www.w3.org/XML/`
[7] JavaScript Object Notation: `https://www.json.org`

```json
{
    "books": [
        {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
        },
        {
         "id": "07",
         "language": "C++",
         "edition": "second",
         "author": "E. Sepp",
         "price": 22.25
        },
    ]
}
```

(a)

```xml
<?xml version="1.0"?>
<books>
    <book>
        <id>01</id>
        <language>Java</language>
        <author>H. Javeson</author>
        <year>2015</year>
    </book>
    <book>
        <id>07</id>
        <language>C++</language>
        <edition>second</edition>
        <author>E. Sepp</author>
        <price>22.25</price>
    </book>
</books>
```

(b)

Figure 4: A JSON and XML representation of the same object.

One should think of these data formats as just that — formats. They are all ways of *serializing* the data structures used in applications into a form that can be transmitted over a network and used by client-side code to implement application logic. XML and Java "grew up together" in the early days of the public internet as the dominant data serialization protocol and application programming language, respectively. It was very easy to convert Java objects into XML documents that could be sent over a network and turned back into Java objects on the other side. JSON has grown in popularity along with the JavaScript programming language over the course of the past two decades, and is the predominant data format used by web applications. Protocol Buffers were developed by Google to meet their internal needs for a data serialization format that was smaller and more efficient than XML.

These data formats were originally developed as ways of exchanging data, but are increasingly being used as the data model for document databases. Microsoft's SQL server supports XML-valued relations; Postgres supports XML and JSON as attribute types; Google's Dremel supports storing Protobuf; CouchBase, MongoDB, Snowflake and many others support JSON.

5.1 JSON Overview

JSON is a text format designed for human-readable and machine-readable interchange of *semi-structured* data. Originally developed to support exchange of data between a web server and Javascript running in a client's browser, it is now used as a native representation within many NoSQL databases. Due to its widespread adoption, we will focus on JSON in this class. JSON supports the following types:

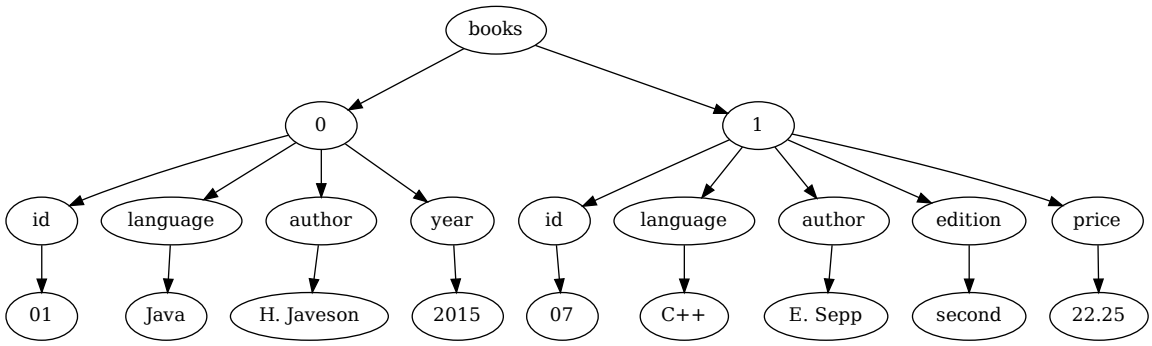


Figure 5: The JSON document in Figure 4 represented as a tree

- **Object**: collection of key-value pairs. Keys must be strings. Values can be any JSON type (i.e., atomic, object, or array). Objects should not contain duplicate keys. Objects are denoted using "{" and "}" in a JSON document.
- **Array**: an *ordered* list of values. Values can be any JSON type (i.e. atomic, object, or array). Arrays are denoted using "[" and "]" in a JSON document.
- **Atomic**: one of a number (a 64-bit float), string, boolean or `null`.

A JSON document consists of one of the above types, which may in turn include nested components. For example, the JSON document in Figure 4 consists of an object with a single key (`"books"`); the value associated with this key is an array of objects. A JSON document can be interpreted as a tree (Figure 5); this is due to the nested structure of JSON documents.

Another important characteristic of JSON is that it is *self-describing* — that is, the schema elements are part of the data itself. This allows each document to have its own schema, even within the same document database.

5.2 JSON vs Relational

Now we can more directly compare document data model (as typified by JSON) with the relational data model.

- **Flexibility**: JSON is a very flexible data model that can easily represent complex structures, including arbitrarily nested data. The relational model is less flexible; it requires keys and joins to represent complex structures.

- **Schema Enforcement**: JSON documents are self-describing; each document in a collection can potentially have a unique structure. Under the relational model, the schema for a table is fixed and all tuples in the table must adhere to the pre-defined schema.
- **Representation**: JSON is a text-based representation, appropriate for exchange over the web. While it is not necessarily the most efficient representation, it can be easily parsed and manipulated by many different applications and programming languages. In contrast, relational data uses a binary representation which is specialized for a particular database system implementation. It is designed for efficient storage and retrieval from disk — not for sharing information. For instance, MySQL and Postgres each have their own binary data format that are mutually incompatible.

  Note that there are intermediate representations such as TSON (typed-JSON) and BSON (binary-JSON) which can enforce schema restrictions or have binary representations that are more efficient than standard JSON; we omit them for this discussion.

As always, there are tradeoffs to be made. We review two of them here.

First, relational databases trade *flexibility* for *simplifying application code*. Relational databases are sometimes described as "enforcing schema on write", and document databases are sometimes described as "enforcing schema on read". Applications querying a relational database will know exactly the structure and data types of the rows that are returned; this can be determined through examining the schemas of the tables being queried. However, applications querying a document database will need to inspect the schema of each document in a collection, which pushes complexity into the application logic. We will investigate this in more detail in Section 5.4 below.

Second, the document model is more suitable for data which is primarily accessed by a *primary key* (like an email or user ID). For these kinds of lookups, the document model exhibits better *locality*; all the data related to the key of interest can be returned in a single query. This is typically more efficient for a document database than for a relational database: information could be stored in multiple tables and require several joins in order to retrieve the same information.

5.3 Converting Between JSON and Relational

Some relational databases such as Postgres and SQLite permit the storage and querying of JSON in a relational context; a table attribute can have a type of `JSON` and elements of the JSON document can be retrieved or queried through the use of special operators and functions. It is also possible to transform JSON documents into a form that can be represented in the relational model, and vice versa.

Transforming Relational to JSON: A single table with a set of attributes can be represented in JSON as an object whose key is the name of the table and whose value is an array of objects. Each object in the array corresponds to a row in the table: the object has the table attributes as keys, and the tuple elements as values. A table with foreign keys into another table (a one-to-many relationship) can be represented in JSON as a nested structure for which the linked rows in the second table are embedded or "inlined" into the objects representing the row of the first table.

```
{
    "person": [
        {"name": "Johnny", "phone": [1234], "address": "123 Maple St., Berkeley, CA", "email": "johnny@example.com"},
        {"name": "Lisa", "phone": [1234, 5678], "address": {
            "street": "456 Oak Ave.", "city": "Berkeley", "state": "CA"
        }}
    ]
}
```

Figure 6: A JSON document which is difficult to translate to a relational table

Many-to-many relationships in a relational schema are harder to represent without introducing duplicate data. Consider a set of three relations, one of which has foreign keys into two other primary-keyed tables (this is the relationship table). There are three options for representing this setup using JSON. The first option is to represent each relation as a flat JSON array (where each element is an object representing a row); this has no redundant data, but relies on the application to join the tables to get related data. The second and third options key off of the one of the primary-keyed tables and inlines the contents of the relationship table and the last table within each object. In either of these cases, the elements of the last table will be duplicated for each linked element of the first table. This redundancy means that the application does not need to perform any joins to get the full record (because it is all stored in the same document), but also means that updates to the document store will need to handle updating all duplicated rows.

Transforming JSON to Relational: Representing semi-structured data as a relation can be tricky because of the potential variation in structure of the documents at hand. Consider the document in Figure 6. This suggests a relational table `person(name, phone, address, email)`, but there are several issues. First, one of the objects in our JSON document has two phone numbers; the schema would need to factor phone numbers out into another table, or suffer the inclusion of duplicate data. Second, one of the objects has an email but another does not. The relational schema can represent the absence of an email with a `NULL` value in the table, but what if future documents contain additional attributes? The schema of the table would need to be changed, which is a potentially expensive manual process. Lastly, the address fields of the two objects have different structures: one contains a simple string and the other contains a nested object. There is no obvious "best" way to represent these two structures in a relational database.

5.4 Querying Semi-Structured Data

Semi-structured data models have their own set of query languages, distinct from SQL. These query languages must handle the variety of structures found in semi-structured data: repeated attributes, different types for the same attributes, nested and heterogeneous collections, and so on. We studied one such language in class (the MongoDB query language). See class lecture slides or the manual[8] for more details.

[8] https://docs.mongodb.com/manual/tutorial/query-documents/

5.5 Problems

1. Are the following workloads better characterized as Online Transaction Processing (OLTP) or Online Analytical Processing (OLAP)?

   (a) Placing orders and buying items in an online marketplace

   (b) Analyzing trends in purchasing habits in an online marketplace

2. Earlier in this class, we learned about the Two Phase Commit protocol. In relation to the CAP theorem, which of the CAP properties (consistency, availability, partition tolerance) does 2PC choose, and which does it give up?

3. Suppose we would like to translate the following data from the relational model to JSON in migrating to a NoSQL database like MongoDB. Given that we want to be able to list all purchases a user made without performing any joins or aggregations, what will the resulting JSON document look like?

Users

userid	username
1	"alice"
2	"bob"

Purchases

userid	productid	cost
1	1	100
1	2	200

5.6 Solutions

1. (a) **OLTP**: OLTP workloads involve high numbers of transactions executed by many different users. Queries in these workloads involve simple lookups more often than complex joins. In this case, when a user buys an item, the site might perform actions like looking up the item and updating its quantity, recording a new purchase, etc.

   (b) **OLAP**: OLAP workloads involve read-only queries and typically include lots of joins and aggregations. Often, workloads executed for analysis and decision making are OLAP workloads.

2. 2PC chooses consistency and partition tolerance, and gives up on availability. By enforcing that a distributed transaction either commits or aborts on all nodes involved, it guarantees that these nodes will provide consistent views of the data. However, we cannot provide all three of the CAP properties, and we give up on availability. For example, consider what

happens to participants who voted yes if the coordinator crashes - they will be stuck in a state of limbo, waiting to learn about the status of the transaction and holding their locks under strict 2PL. If other transactions try to access the data these locks are held on, they will end up on a wait queue; hence, this data is unavailable.

3. To avoid performing joins when finding all purchases for a user, we can inline Purchases within Users, storing a list of purchases for each user.

```
{
  "users": [
    {
      "userid": 1,
      "name": "Alice",
      "purchases": [
        {
          "productid": 1,
          "cost": 100
        },
        {
          "productid": 2,
          "cost": 200
        }
      ]
    },
    {
      "userid": 2,
      "name": "Bob",
      "purchases": []
    }
  ]
}
```

CS 186
Spring 2021

MapReduce and Spark

1 Introduction

In previous modules, we learned how to parallelize relational database systems which is useful for optimizing data processing but only works well up to a certain number of machines. Difficulties and headaches come when we start thinking about scaling the relational model on databases split across hundreds or thousands of machines. This became a problem when more and more people, especially every day consumers, started to interact with databases when the Internet exploded around the turn of the 21st century. Database thinking, models, and technologies needed to catch up to meet the demand. This note will focus on two recent advances in parallel database technologies - MapReduce and Spark - that have enabled the massive scalability of modern data processing.

2 MapReduce

2.1 DFS and High-Level Introduction

Engineers at Google in the early 2000s needed a way to efficiently manage and process the large amounts of data (at petabyte scale and beyond) that was being generated, stored, and indexed by the company. First off, they needed a way to store files across hundreds and eventually thousands of machines since only a few would definitely not be enough. To do this, they designed a file system in which large files (TBs, PBs) are partitioned into smaller files called **chunks** (usually 64MB) and then distributed and replicated several times on different nodes for fault tolerance. This is called a **distributed file system (DFS)** which has had countless implementations since then from Google's proprietary GFS to Hadoop's open source HDFS.

Next, they needed a way to efficiently process the large amount of data stored on a DFS. To do this, they built MapReduce which is a high-level programming model and implementation for large-scale parallel data processing. Its name is derived from the two main and separate phases of the process: Map and Reduce. From a high level, we can describe the Map phase as applying a function in parallel to every element of a set of data and the Reduce phase as combining the results of the Map phase into the desired data output. The magic of this paradigm in processing data at scale is it automatically handles the details of issuing and managing tasks in parallel across multiple machines. A user only has to define the Map and Reduce tasks and MapReduce takes care of the rest, similar to how SQL creates an execution plan from a query.

2.2 Data Model and Programming

The data model in MapReduce works on files are bags `(key, value)` pairs. A MapReduce program takes in an input of a bag of `(input_key, value)` pairs and outputs a bag of `(output_key, value)` pairs (`output_key` is optional). The user must provide two stateless functions – `map()` and `reduce()` – to define how the input pairs will be transformed into the output pair domain.

The first part of MapReduce – the Map phase – applies a user-provided Map function in parallel to an input of `(key1, value1)` pairs and outputs a bag of `(key2, value2)` pairs. The types and values of `key1`, `value1`, `key2`, and `value2` are independent of each other. The `(key2, value2)` pairs serve as intermediate tuples in the MapReduce process.

```
func map(key1, value1) -> bag((key2, value2))
```

The second part of MapReduce – the Reduce phase – groups all intermediate pairs outputted by the Map phase with the same `key2` and processes each group into a single bag of output values defined by a user-provided Reduce function. This function is also run in parallel with a machine or worker thread processing a single group of pairs with the same `key2`. The `(key3, value3)` pairs in this function serve as the output tuples in the MapReduce process.

```
func reduce(key2, bag(value2)) -> bag((key3, value3))
```

2.3 Word Count Example

Let's start with an example of counting the number of occurrences of each word in a large collection of documents. Each document is a bag of key-value pairs `(did, word)` with key `did` being the document `did` and value `word` being the entire set of words in that document.

We define the following map function to emit an intermediate `(w, 1)` pair for every word that appears in a document. Note that this function is stateless so we can run it in parallel.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        emitIntermediate(w, 1)
```

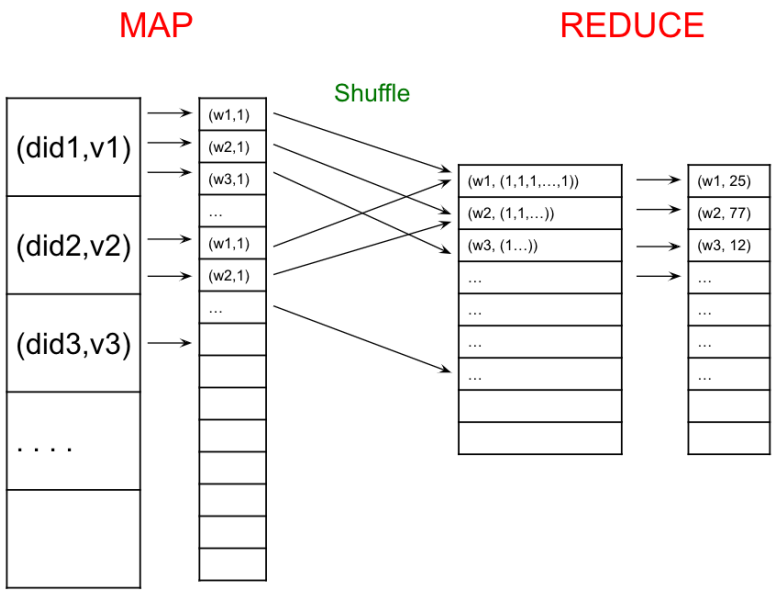
After applying this map function to all documents, we essentially have a collection of all individual words that appear in all documents with a 1 assigned to each to represent the number of appearances. For example:

```
(apple, 1)
(banana, 1)
(banana, 1)
(book, 1)
...
(shoe, 1)
```

MapReduce then groups intermediate pairs with the same word key and for each group, combines their values (in this case all 1s) into an iterator. The following reduce function takes the iterator for each word key and returns the sum over the 1s in the values, resulting in the count of each word across all documents.

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int sum = 0;
    for each v in values:
        sum += v;
    emit(key, sum);
```

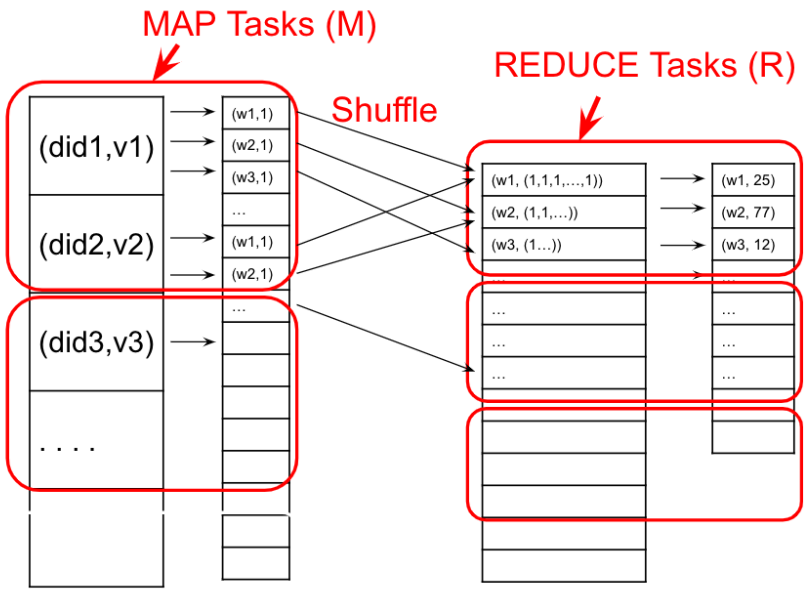
Here is a visual diagram of what happens at a high level in MapReduce. First, we map each document to a list of (word, 1) pairs. Then, we shuffle (or group) these intermediate pairs based on the word and create a list of 1s for each group. Final, the reduce function sums over this list to generate the word count for each word.



2.4 Workers

MapReduce processes are split up and run by **workers** which are processes that execute one task at a time. With 1 worker per core, we can have multiple workers per node in our system.

Here is the same Word Count example but with worker tasks denoted by what part of the MapReduce process they handle. Both the Map and Reduce phases can be split among different workers on different machines, with workers performing independent tasks in parallel. Also, the shuffling process here is automatically handled by the system.



2.5 Implementation

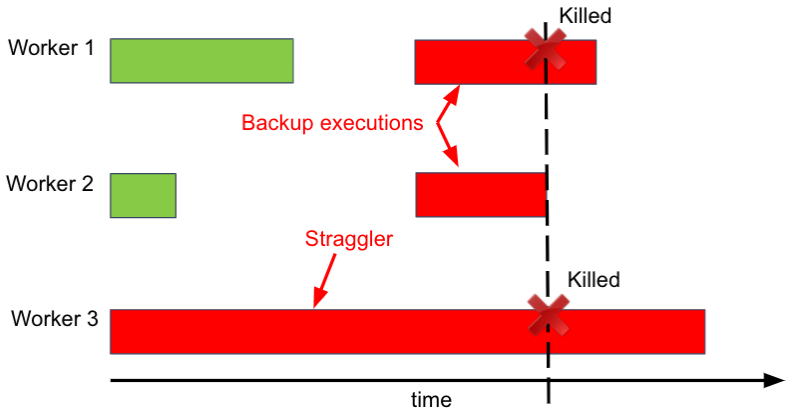
Like 2PC, MapReduce designates one machine as the leader node. Its role is to accept new MapReduce tasks and assign them to workers later on. The leader begins by partitioning the input file into M splits by key and assigning workers to M map tasks, keeping track of progress as workers perform their tasks. Workers write their output to the local disk and partition their output into R regions. The leader then assigns workers to the R reduce tasks which then write the final output to disk once complete.

2.6 Fault Tolerance

Now, we can start thinking of possibilities of failure in this implementation. One of the most common faults is the failure of a worker performing a map task. One way to solve this is to just restart a new mapper task and read the data it is supposed to process from the disk again. However, since the Reduce phase must wait until the entire Map phase completes, does this mean if one map task fails then we must restart the entire Map phase again?

No. If we write the intermediate result of the Map phase to disk, then we can avoid this. This means that the Reduce phase workers will have to read files from disk as its input and exactly like the Map phase workers - if one crashes during execution, it just gets restarted.

Another interesting detail about the MapReduce implementation is how it handles machines that take an unusually long time to complete one of its tasks, also known as a **straggler**. These stragglers could be the result of faulty hardware, too many tasks assigned to one machine, or any number of other reasons. To avoid this, MapReduce preemptively starts new executions of the last few remaining in-progress tasks, with the hope that all the necessary tasks are completed.



For example, in this case of 3 workers, worker 1 and 2 complete their tasks first while worker 3 straggles. MapReduce then decides to utilize worker 1 and 2, which are now idle, to start the same task as worker 3 as backup. Once worker 2 finishes this task, the remaining tasks on worker 1 and 3 are killed and the MapReduce process can proceed.

2.7 Selection, Group By, Join

Let's look at how to implement a selection (from relational algebra) using MapReduce. If we want to perform a query like $\sigma_{A=123}(R)$, our map and reduce functions are:

```
map(Tuple t):
    if t.A = 123:
        EmitIntermediate(t.A, t);

reduce(String A, Iterator values):
    for each v in values:
        Emit(v);
```

The map function is applied over all the data and only outputs tuples if their A field = 123. The reduce function just serves to output the tuples in the end. While the reduce function is arguably unnecessary in this case, we must always supply both the map and reduce functions to MapReduce.

Moving on, to implement a group by operator such as $\gamma_{A,sum(B)}(R)$, our map and reduce functions are:

```
map(Tuple t):
    EmitIntermediate(t.A, t.B);

reduce(String A, Iterator values):
    s = 0;
    for each v in values:
        s = s + v;
    Emit(A, s);
```

In the map function, we assign the key of the intermediate pairs as `t.A` since we are grouping by A and assign the value as `t.B` since we want to sum over the B values. In the reduce function, we do just that as we iterate over the B values of each of the group.

Finally, let's tackle the more challenging yet still practical implementation of a inner join using MapReduce. If we have relations $R(A, B)$ and $S(C, D)$ and want to perform the inner join $R \bowtie_{B=C} S$, we need to be a bit more clever in how we structure our map and reduce functions. For our map function, we need to output intermediate pairs such that tuples from R will be matched with tuples from S on R.B = S.C and whose source table is stored. These pairs will be grouped by the matching R.B = S.C values and will then need to be processed by a reduce function which can separate tuples by their source tables, create two separate lists, and output each pair from the two lists. These map and reduce functions do just that:

```
map(Tuple t):
    switch (t.relationName):
    case 'R': EmitIntermediate(t.B, t);
    case 'S': EmitIntermediate(t.C, t);

reduce(String k, Iterator values):
    R = []; S = [];
    for each v in values:
        switch (v.relationName):
        case 'R': R.append(v)
        case 'S': S.append(v);
    for v1 in R, v2 in S:
        Emit(v1, v2);
```

3 Spark

3.1 History and Motivation

We've seen how MapReduce provides a high-level programming paradigm to handling scalable, distributed, fault-tolerant data processes across many machines. However, it comes with several drawbacks and limitations:

- As mentioned previously, users must always provide both the map and the reduce functions. With such a strict format, it can be difficult to write more complex queries.
- Writing intermediate results to disk is necessary for fault tolerance, but can be very slow.
- Running multiple MapReduce jobs can take a long time due to waiting for writes to finish.

Spark was developed right here in Berkeley to address these issues. It is an open source system that handles distributed processing over HDFS like MapReduce. But unlike MapReduce, Spark:

- Consists of multiple steps instead of 1 mapper + 1 reducer.
- Stores intermediate results in main memory.
- Resembles relational algebra more closely.

3.2 Resilient Distributed Datasets

Spark's data model consists of semi-structured data objects called **Resilient Distributed Datasets (RDDs)**. These objects, which can be anything from key value pairs to objects of a certain type, are immutable datasets that can be distributed across multiple machines. For faster execution, RDDs are not written to disk in intermediate steps but are rather stored in main memory. Since this will lead to intermediate results being lost if a node crashes, each RDD's **lineage** is tracked which can help recompute RDDs. This means that we must store the lineage in persistent storage, like writing to disk before executing the program, to generate the output RDD. In practice, this can be done by writing to a log and flushing it to the disk, just like what we learned earlier in the Recovery module.

3.3 Programming

Spark programs consist of two types of operators: **transformations** and **actions**. Actions are **eager** operators which means they are executed immediately when called. These include operators such as count, reduce, and save. Transformations, on the other hand, are evaluated **lazily**, meaning they are not executed immediately but are rather recorded in the lineage log. An operator tree, similar to a relational algebra tree, is constructed in memory. Transformations were designed this way to optimize operator execution, similar to how SQL optimizes over a set of query plans. For

example, if we have two consecutive selections, we can combine them to a single selection to save 1 entire pass over the RDD which saves I/Os. Other transformations include map, reduceByKey, join, and filter.

Now let's see how Spark is actually programmed. In this example, we are reading a large log as a textfile and want to output all lines that start "NullPointerException" and contain "SQLite":

```
s = SparkSession.builder()...getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
nullErrors = lines.filter(l -> l.startsWith("NullPointerException"));
sqlLiteErrors = nullErrors.filter(l -> l.contains("SQLite"));
sqlLiteErrors.collect();
```

Here, the first line creates a context object that can be used to create RDDs. The second line then reads the log. Filtering on the two conditions is performed by the third and fourth line, but remember this is a transformation so they are not evaluated until we call `collect()` on the last line which is an action and thus, triggers the computation.

A more concise and arguably cleaner version of this Spark code which better illustrates data flow can be written like this ("call chaining" style):

```
s = SparkSession.builder()...getOrCreate();
errors = s.read().textFile("hdfs://logfile.log")
    .filter(l -> l.startsWith("NullPointerException"))
    .filter(l -> l.contains("SQLite"))
    .collect();
```

3.4 Persistence

Since Spark does not write intermediate results to disk, if a server fails before a program finishes, the entire execution of that program must restart. However, Spark does allow options to avoid that. If we want to persist, or materialize, our intermediate results, we can call `persist()` at points in our code like this:

```
s = SparkSession.builder()...getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
nullErrors = lines.filter(l -> l.startsWith("NullPointerException"));
nullErrors.persist(); \\ <-- MATERIALIZATION
sqlLiteErrors = nullErrors.filter(l -> l.contains("SQLite"));
sqlLiteErrors.collect();
```

This will fully compute `nullErrors` as an RDD and store the results to disk, essentially creating a checkpoint at which we can restart if the server fails.

3.5 Join Example

Let's convert the following SQL query into Spark code:

```
SELECT COUNT(*) FROM R, S
WHERE R.B > 200 AND S.C < 100 AND R.A = S.A
```

We can read in R and S and parse each line into an object, persisting on disk this intermediate result like this:

```
R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.read().textFile("S.csv").map(parseRecord).persist();
```

Then we create our transformations to apply the two single relation conditions in the where clause:

```
RB = R.filter(t -> t.b > 200).persist();
SC = S.filter(t -> t.c < 100).persist();
```

To join the two filtered relations, we can simply use Spark's `join()` function:

```
J = RB.join(SC).persist();
```

Finally, we apply the `count` like so:

```
J.count();
```

3.6 A Partial List of Spark Transformations and Actions

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

3.7 Spark 2.0 – DataFrames and Datasets

Spark 2.0 introduced two new forms of data collections: DataFrames and Datasets.

DataFrames are very similar to relations in that each object in a DataFrame is called a **Row** and have attributes called **Columns**. They also have more SQL-like API functions such as `agg()` and `col()`. Let's look at converting the following SQL query to Spark using Datasets (source):

```
SELECT AVG(p.salary), MAX(p.age)
FROM people p, department d
WHERE p.age > 30 AND p.deptId = d.id
GROUP BY d.name, gender
```

We can read in the `people` and `department` relations and then chain transformations and actions like so:

```
Dataset<Row> people = s.read().textFile("R.csv");
Dataset<Row> department = s.read().textFile("S.csv");

people.filter("age".gt(30))
    .join(department, people.col("deptId").equalTo(department("id")))
    .groupBy(department.col("name"), "gender")
    .agg(avg(people.col("salary")), max(people.col("age")));
```

Datasets are a more general type of DataFrames in which its elements must be typed objects (`Dataset<People>` rather than `Dataset<Row>`). In fact, DataFrames are just Datasets with object type `Row`. Here is a sample of the Datasets API:

```
agg(Column expr, Column... exprs)
Aggregates on the entire Dataset without groups.

groupBy(String col1, String... cols)
Groups the Dataset using the specified columns, so that we can run aggregation on them.

join(Dataset<?> right)
Join with another DataFrame.

orderBy(Column... sortExprs)
Returns a new Dataset sorted by the given expressions.

select(Column... cols)
Selects a set of column based expressions.
```

4 Conclusion

In this note, we looked at two technologies that have been design to perform parallel data processing on the terabyte scale and beyond: MapReduce and Spark. MapReduce came first and utilized user-defined map and reduce functions (hence its name). It writes intermediate results to disk which is great for fault tolerance but bad for I/O cost and performance. It also pioneered the model of automatically assigning, tracking, and managing tasks among workers across many machines and is equipped to handle failures and stragglers.

Spark came along to address some of the issues of MapReduce which include its strict structure, difficulty in writing complex queries, and slow performance due to intermediate writes. Spark uses the object-based Resilient Distributed Dataset (RDD) and is operated by transformations, which are evaluated lazily, and actions, which are evaluated eagerly. Spark more closely resembles relational algebra, has features that allow for persistence, and recently introduced new data models (DataFrames, Datasets) which better resembles the relational data model.