Databases

Muchang Bahng

Fall 2024

Contents

1 Index and B+ Trees

So far, we've abstracted away the inner workings of the DBMS and was just told to trust it. From here, we'll delve into the inner workings of modern DBMS systems, starting with hardware.

1.1 Hardware and Memory Layout

Recall the memory hierarchy, which starts with CPU registers, followed by caches, memory, and a disk. The speed of read/write, called I/O, determines how fast you can retrieve this data.

Definition 1.1 (Hard Disk)

In SQL queries, the (disk) I/O dominates the execution time, so this must be analyzed first. A typical hard drive consists of a bunch of **disks/platters** that are spun by a **spindle** and read by a **disk head** on a **disk arm**. Each disk is pizza sliced into **sectors** and donut sliced into **tracks** (and a collection of tracks over all disks is a **cylinder**), and the arm will read one **block** of information, which is a logical unit of transfer consisting of one or more blocks (i.e. like a word, which is usually 4 bytes).[a]

Definition 1.2 (Access Time)

The access time is the sum of

1. the *seek time*: time for disk heads to move to the correct cylinder
2. the *rotational delay*: time for the desired block to rotate under the disk head
3. the *transfer time*: time to read/write data on the block

This spindle rotation and moving arm is slow, so the times are dominated by the first two.

Note that this is heavily dependent on the data being accessed. Sequential data is extremely fast while random access is slow. Therefore, we should try to store data that should be accessed together next to each other in the disk.

Definition 1.3 (Memory, Buffer Pool)

As we expect, the DBMS stores a cache of blocks, called the **memory/buffer pool** containing some fixed size of M maximum blocks. We read/write to this pool of blocks (which costs some time per blocks) and then these dirty (which are written/updated) are flushed back to the disk. It essentially acts as a middleman between the disk and the programmers, and every piece of data that goes between the two must go through the memory.

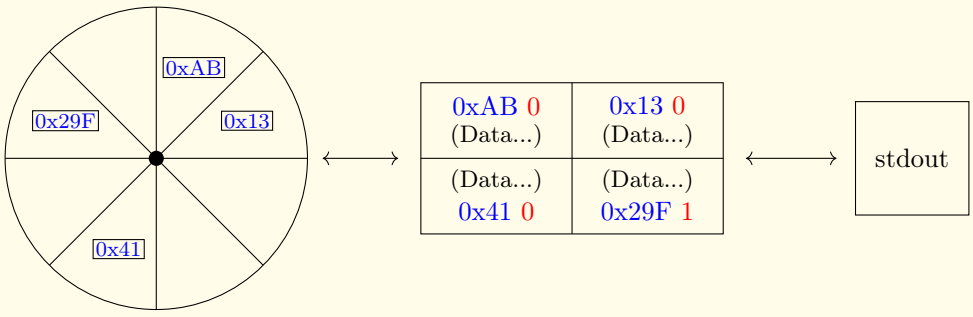


Figure 1: We store the memory address of the block (blue) and a bit indicating whether it is dirty or not (red).

[a] You cannot just write 1 byte. You must rewrite the entire block with the 1 byte updated. If we want to write 2 bytes which are on separate blocks, we must do 2 block writes.

Therefore, if we want to read $N > M$ blocks, then the first M blocks must be loaded into memory, outputted into stdout, and then the memory must be refilled with the rest of the blocks. If we have updated a block in memory, then we should flush it before overwriting this block in memory.[1] Replacement strategies won't be covered here. Note that unlike algorithms, which focus on the cost of the algorithm after it is read into memory, we focus on the cost of loading data from the disk into memory.

So how should we increase performance?

1. *Disk Layout*: Keep related things close together, e.g. in the same sector/block, or same track, or same cylinder, or adjacent cylinders.
2. *Prefetching*: When fetching a block from the disk, fetch the next block as well since it's pretty likely to access data from the next block. This is basically locality.
3. *Parallel I/O*: We can have more heads working at the same time.
4. *Disk Scheduling Algorithm*: e.g. the elevator algorithm sorts the cylinders so that you don't go back and forth between cylinders when fetching.
5. *Track Buffer*: Read/write one entire track at a time.

Now let's talk about how the actual bytes are stored in memory.

Definition 1.4 (Row Major Order, NSM)

We group rows together contiguously in the disk block.[a] If we have a relation with schema `R(INT(4), CHAR(24), INT(4), DOUBLE(8))`, we first store the rows together, with extra space in between since we might append new attributes, and have a tuple of pointers to the start of each row (orange lines).

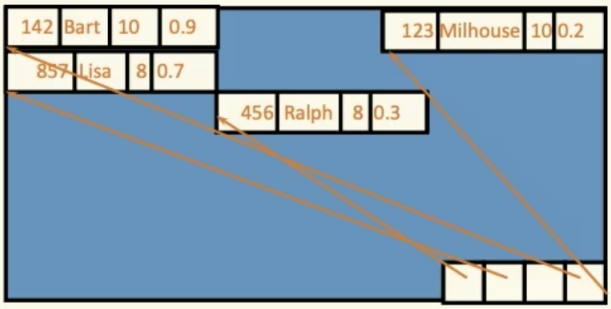


Figure 2

Note that `VARCHAR` still allocates the same number of bytes, but adds padding.

Definition 1.5 (Column Major, PAX)

We group columns together contiguously in the disk block, which allows for optimization since all types are the same and we can just use pointer arithmetic to get every attribute in $O(1)$ time. We split the block into chunks and have metadata of pointers that point to the start of the array for each attribute.

[1] idk perhaps we can have a overhead bit indicating whether a block is updated, along with the memory address of this block so it can find it back on the disk.

[a] This is the most standard storage policy.

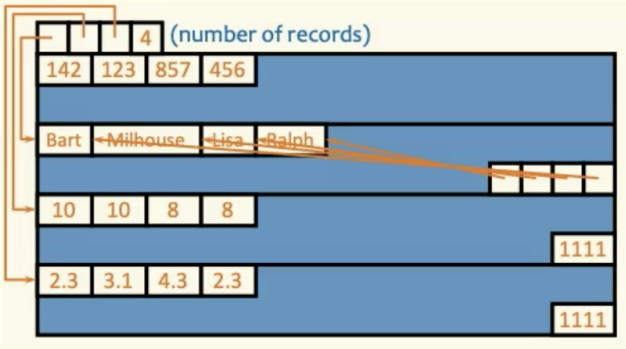


Figure 3

1.2 Search Keys and Index

Now that we've seen how the data is actually laid out in memory, we can look under the hood to see what happens when we make a query as such. Consider the relation schema `User(uid, age)`, along with the two SQL queries, specifying on the key attribute and a non-key attribute.

```sql
SELECT *
FROM User
WHERE uid = 112;
```

```sql
SELECT *
FROM User
WHERE age = 12;
```

The DBMS will have to go to disk and scan it to find the tuples satisfying the predicate. However, it is a bit more sophisticated than a simple complete scan.

Definition 1.6 (Search Key)

The **search key** is simply the attribute on which our query is defined (e.g. `uid` and `age` in the example above). The values (e.g. `112` and `12`) are called the **search key values**.

This is distinct from a closely-related concept called the index, which also refers to attributes. The tuples may take on some structure depending on the attribute. It may be sorted in one attribute but random in another.

Definition 1.7 (Index)

This set of attribute values, which really act as pointers, that we use to scan more efficiently on disk is called the **index**.

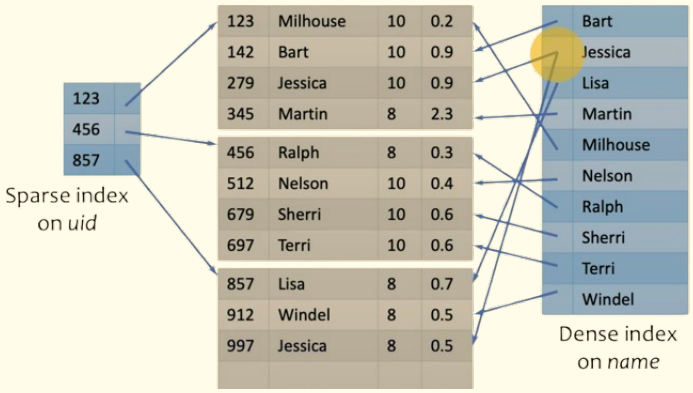


Figure 4: The dense index has 10 values, but there are two Jessicas, so it is indeed dense. The sparse index is on the uid, and since the relation is sorted (a more specific form of *clustering*) in the disk, we can use sparse keys.

Sparse and dense just refers to how much an index "covers" a disk block.

1. A **dense index** means that there is one index entry for each search key value. One entry may point to multiple records
2. A **sparse index** means that there is one index entry for each block. Records must be clustered according to the search key as shown below, and this can optimize searching.

Note that

1. The sparse index must contain at least the number of blocks, while the dense index must contain at least the number of unique search key values. Since the sparse index is much smaller, we may be able to fit it into main memory and not have to pay the I/O cost.
2. A dense index does not require anything on the records, while the sparse requires the data to be clustered.
3. Lookup is easy on dense since we can directly see if it exists. For sparse, we must first follow the pointer and scan the block. (e.g. if we wanted to look for 279, we want to look at the address pointed to by 123, and scan down until we hit it or reach a number greater than 279).
4. Update is usually easier on sparse indices since we don't need to update the index unless we add a new block. For dense, if we added a new person Muchang, then we would have to add `Muchang` to the dense index.

Definition 1.8 (Primary vs Secondary Index)

Primary and secondary refers to what the index is over.

1. A **primary index** is created for the primary key of the relation. Records are usually clustered by the primary key, so this can be sparse.
2. A **secondary index** is any index that is not over a primary key and is usually dense.

In SQL, the `PRIMARY KEY` declaration automatically creates a primary index, and the `UNIQUE` declaration creates a secondary index.

Example 1.1 (Additional Secondary Indices)

You can also create an additional secondary index on non-key attributes. For example, if you think that you will query based on popularity often, you can do

```
CREATE INDEX UserPopIndex ON User(pop);
```

which will create a dense index to speed up lookup at the cost of memory.

1.3 Tree Index

1.3.1 B-Trees

So really, an index is really a set of pointers, but we must be able to store this set of pointers. This leads to the problem of the index being too big to fit into a block. In this case, we can just create a sparse index on top of the dense index. This allows us to store a large index across blocks.

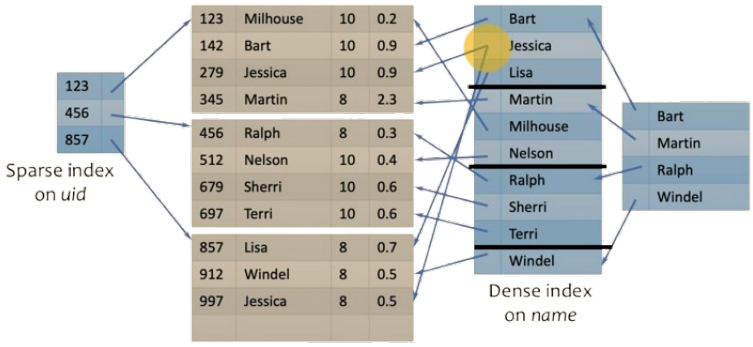


Figure 5: We store a sparse index in memory which points to a dense index on disk, which then points to the tuples of the relation, also on disk.

If the index is still too big, we store another index on top of that, and we have pretty much a tree. This is called the Index Sequential Access Method (ISAM).

Example 1.2 (Index Traversal in Tree)

If we want to look up 197 in this tree, we traverse down. Note that we have the root node in memory already, but we would require an IO operation to traverse the pointer to the first block, and then another IO cost to go to the second.

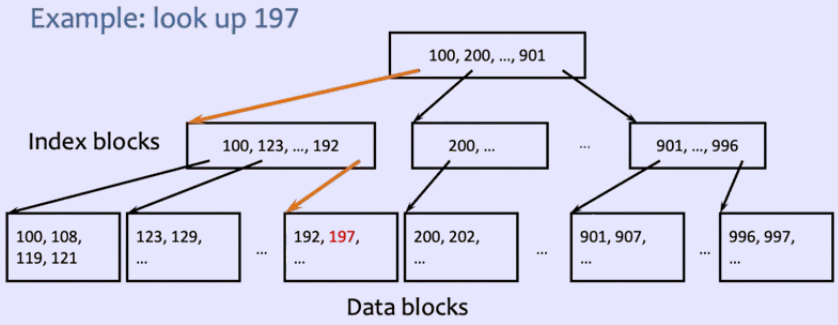


Figure 6: An index tree of depth 3.

A problem with this method is also a problem with BSTs. If we want to delete 123 and add 107 ten times, then we have an unbalanced binary search tree and in extreme cases, this reduces to a linear search.

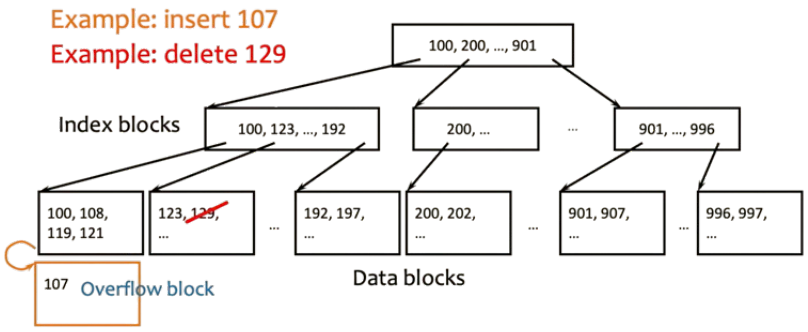


Figure 7: If this block overflows, then we want to expand this, leading to an unbalanced BST and reducing our search to linear time.

Definition 1.9 (B Tree)

Therefore, this static data structure is not good, and we must use a more flexible one, called a **B-tree**.

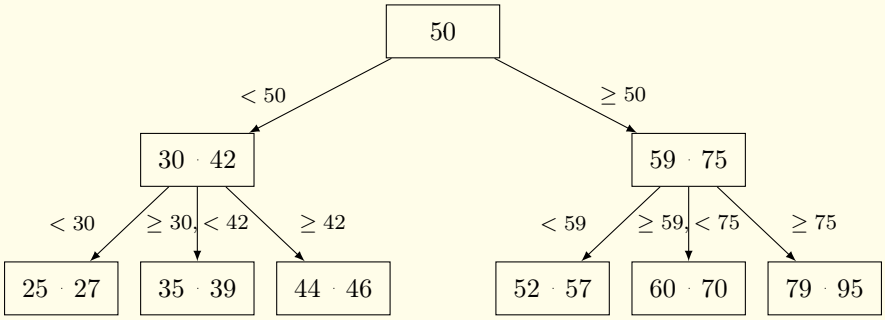


Figure 8: B+ tree structure where each node can hold multiple entries with fixed maximum size and is sorted. The tree maintains height balance, and all leaf nodes are sorted. Each node does not need to be full, and the number of pointers equals the number of entries plus one.

1.3.2 B+ Trees

The actual structure that modern DBMS uses is the slightly more sophisticated B+ tree.

Definition 1.10 (B+ Tree)

While in B trees, the non-leaf nodes are also data, **B+ trees** divide the nodes into leaf nodes, which represent the data in this data structure, and the non-leaf nodes, which do not represent data but index nodes containing index entries.

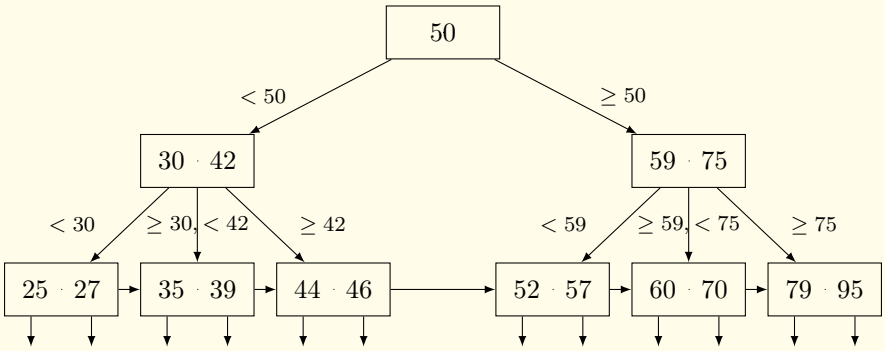
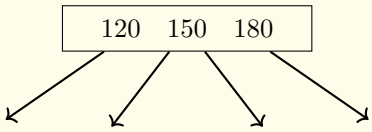


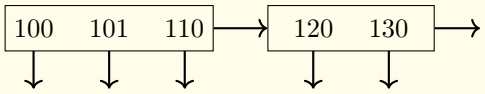
Figure 9: The main difference is that the index nodes contain index entries, and the leaves are linked. Note that the 59, which represents both the index and the data value, are repeated. In here, we assume a block size of 2. The leaf nodes are the indices that point to the tuples in the disk/memory. Sometimes, we may store the tuple directly in the nodes, which saves us another level of indirection, but this may cause memory problems if the tuples have too many attributes.

It has the following constraints.

1. The **fanout** refers the to maximum number of pointers that can come out of each node.



(a) Non-leaf node of a B-tree with four child pointers and three separator keys



(b) Leaf nodes of a B-tree with pointers to data and links to adjacent leaves

Figure 10: With fanout constraint 4, for index nodes, we have 3 values with 4 pointers representing each range. For the leaves, we have 3 pointers to the disk addresses plus 1 pointer to the next node.

2. The root is the only node that can store one value. It is special in this way.

Therefore, if we have a fanout of f, the table shows the constraints.

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil$

Figure 11: Note that there is a minimum constraint to ensure that the tree is balanced.

Now let's describe the implementation behind its supported operations.

Algorithm 1.1 (Lookup)

If we query `SELECT * FROM R WHERE k = 179;`, we go through the B+ tree's indices and reach the leaf node. From here, we can use the pointer to go to the memory address holding this tuple in memory/disk. If we query `k = 32`, then we will not find it after reaching `35`. If we want to query a range `32 < k < 179`, we start at 32 and use the leaf pointers to go to the next leaf until we hit 179.

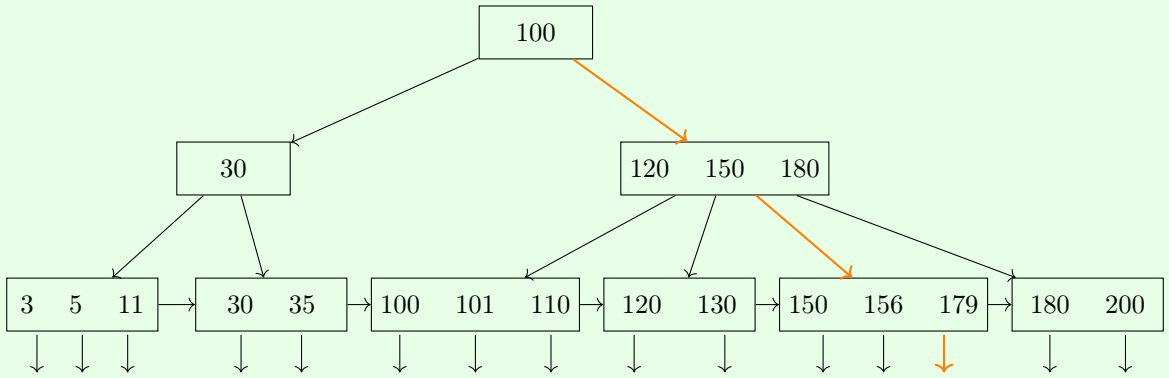


Figure 12: B-tree structure with nodes containing multiple values and pointers to child nodes. Leaf nodes have horizontal connections for sequential access and vertical pointers to data records. Note that the max fanout is 4.

In practice, there are much more pointers, so we could start at 179 and go back to 32 if we had backwards pointers.

Algorithm 1.2 (Insertion)

Insertion onto a leaf node having space is easy.

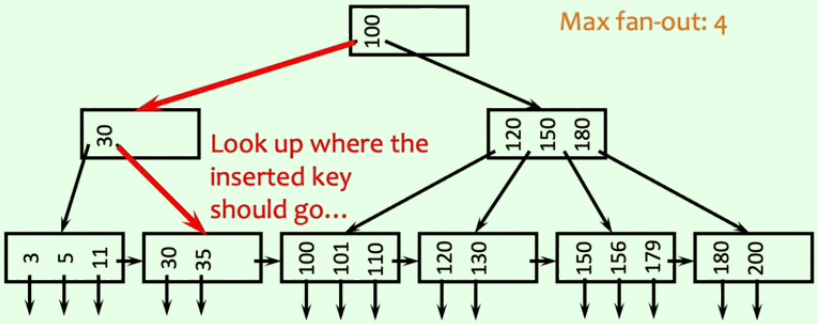


Figure 13: Note that to traverse this tree, we had to access 3 blocks of memory since for each block, we had to go to the disk, look up its contents, and retrieve the index of the next block (retrieve the blocks containing (100), (30), and (30, 35)) Then we have to update this block of (30, 35) and flush it.

When we have a full node, it is more complicated.

1. Say that we have a full node. We can try to push the rest of the leaf nodes to next node having space, but we are not guaranteed that the neighbors will have space.

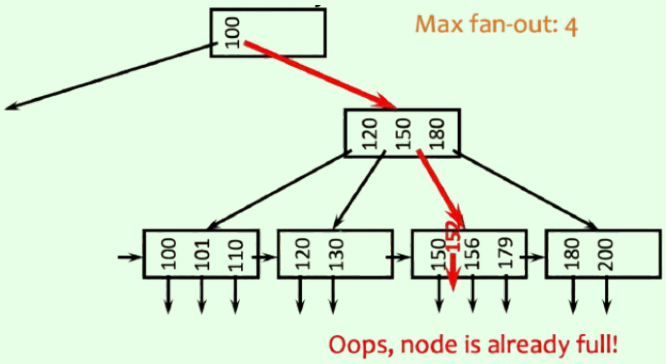


Figure 14

2. We can split the node, called a **copy-up**. However, this new node now has no pointer. If the parent node is not full, we can just add the pointer and update its value.

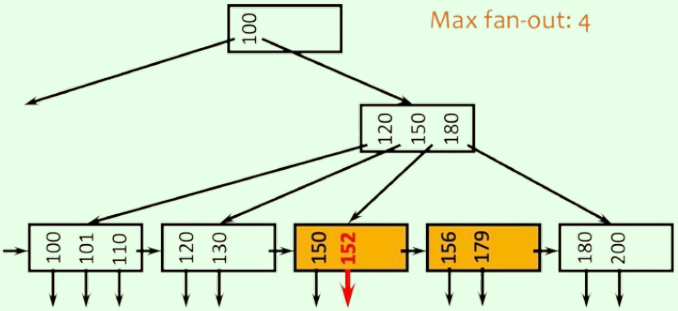


Figure 15

3. If it is full, then we should split the parent node as well. This is called a **push-up**.

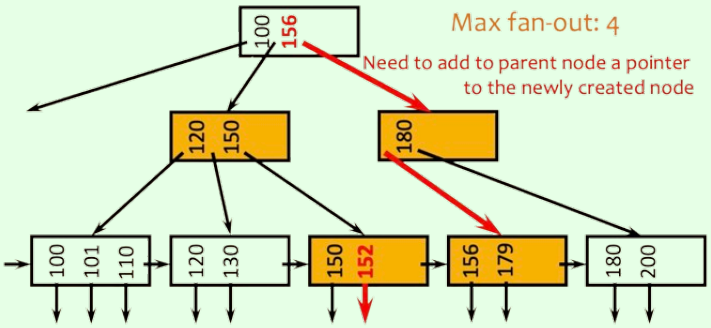


Figure 16

4. This means that we have to update the parent of the parent. If the parent is not full, then we simply add it, and if it is full, then we split the parent of the parent, and so on. If we reach the root node, then we just split the root and increase the height of the tree.[a]

[a] This is rare in practice since the fanout is much greater than 3.

Algorithm 1.3 (Deletion)

Deleting a value is simple if after deletion, the leaf node has at least $f/2$ pointers ($f/2 - 1$ values).

1. If the node has less than $f/2$ pointers, then it is too empty. We can adjust by taking adjacent nodes and moving them from the full nodes to the empty nodes, but this may steal too much from siblings.

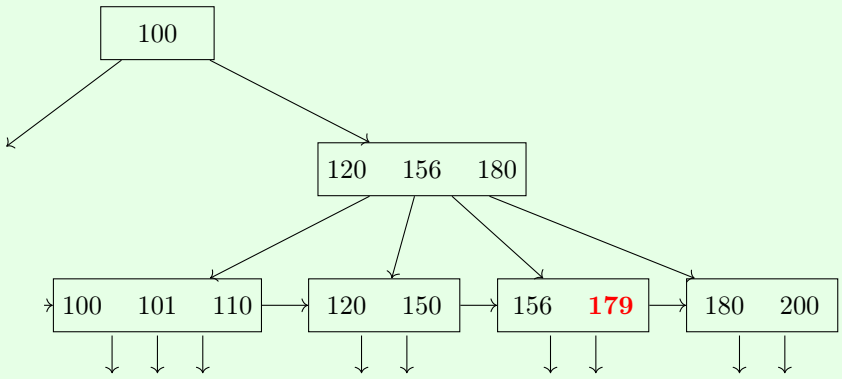


Figure 17: If you try to delete 179 and take any index from the adjacent leaf nodes, they would be too empty as well!

2. The adjacent nodes may be empty as well, and in this case we want to **coalesce** or merge the nodes together. This results in a dangling pointer, so we delete the pointer and remove a value from the parent node.[a]

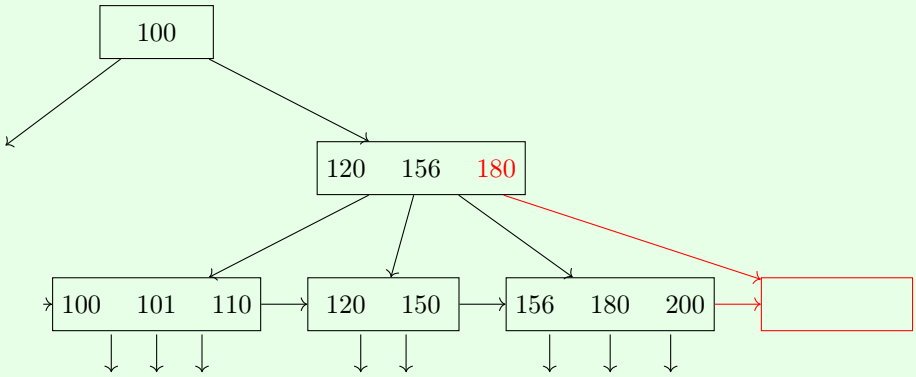


Figure 18: You should merge the right node into the node with the deleted element, and then delete the nodes and pointers that are not needed (in red).

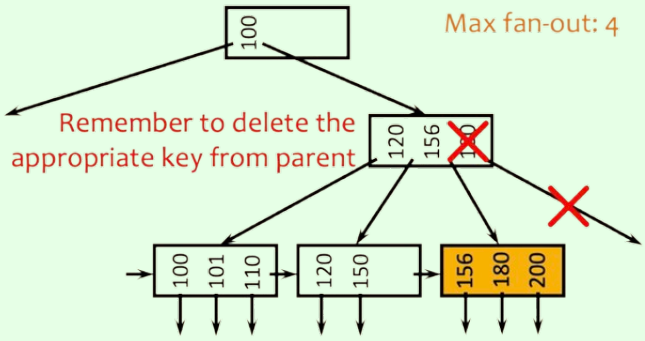


Figure 19

3. We keep doing this until the B+ tree requirements are satisfied or we reach the root, at which point we delete the root node and our B+ tree height decreases by 1.

[a] In practice, this is not done every deletion. This reorganizing process can be deferred and the B+ tree property may temporarily not hold.

Theorem 1.1 (IO Cost of Lookup, Insertion, Deletion)

In general, all these operations have similar runtimes:

1. They require us to go to the bottom of the tree, so it is h operations, where h is the height.
2. We also maybe have $+1$ or $+2$ to manipulate the actual records, plus $O(h)$ for reorganization like we saw before (which is rare if f is large).
3. Minus 1 if we cache the root in memory, which can be decreased even further if we cache more blocks.

The actual size of h is roughly $\log_{f/2} N$, where N is the number of records. f is the fanout, but the B+ tree properties guarantee that there are at least $f/2$ pointers, so it is $\log_{f/2} N$ at worst and $\log_f N$ at best. f is very large usually so this is quite good.

The reason we use B+ trees rather than B trees is that if we store data in non-leaf nodes, this decreases the fanout f and increases h. Therefore, records in leaves require more I/Os to access.

1.3.3 Clustered vs Unclustered Index

Note that in a B+ tree, the leaf nodes always store the index in sorted order. This is so we can have fast lookup in indices. When we go into the disk, however, we may not have this assumption.

Definition 1.11 (Clustered vs Unclustered Index)

If the order of the data records on disk is the same as or close to the order of data entries in an index, then it is **clustered**, and otherwise **unclustered**.

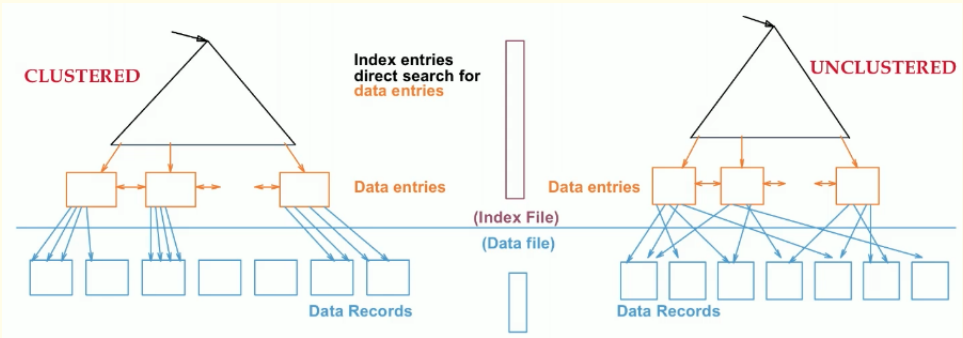


Figure 20: Even if the data entries (leaf nodes) are sorted, the memory addresses of the blocks on disk that they point to may not be sorted.

Note that the B+ tree is still a search tree! It is sorted. The clustered is a property of the data on disk (blue squares).

The performance can really depend on whether the index is clustered or unclustered.

1.4 Hash and Composite Index

Definition 1.12 (Hash Index)

So far, we have used tree indices. However, an alternative is to use a **hash index** which hashes the search keys for comparison. Hash indices can only handle equality queries.

1. `SELECT * FROM R WHERE age = 5;` (requires hash index on `(age)`)
2. `SELECT * FROM R, S WHERE R.A = S.A;` (requires hash index on `R.A` or `S.A`)
3. `SELECT * FROM R WHERE age = 5 AND name = 'Bob'` (requires hash index on `(age, name)`)

They are more amenable to parallel processing but *cannot handle range queries or prefixes*, e.g. `SELECT * FROM R WHERE age >= 5;`. Its performance depends on how good the hash function is (whether it distributes data uniformly and whether data has skew).

Definition 1.13 (Composite Index)

We've looked at queries in the form `SELECT  FROM User WHERE age = 50;`, but what if there are multiple conditions. For example, if we have

```
SELECT * FROM User WHERE age >= 25 AND name = 'B';
```

then we can do a couple things:

1. We have the index on `(age)`, at which point the leaf nodes will look something like

$$25, 25, 25, 26, 26, 28, 29, 29 \tag{1}$$

   We traverse through all the addresses in these leaf nodes and find the ones with name `B`.
2. If we have a **composite index** on `(age, name)`, then our leaf nodes will sort them as

$$(25, A), (25, A), (25, B), (26, A), (26, C), (28, B), (29, A), (29, C) \tag{2}$$

3. If we index on `(name, age)`, then our leaf nodes will sort them as

$$(A, 25), (A, 25), (A, 26), (A, 29), (B, 25), (B, 28), (C, 26), (C, 29) \tag{3}$$

Note that if we had the query `SELECT  FROM R WHERE age >= 25`, then this sorting would not help since we do not prioritize ordering by age. So we cannot use tree indexing over name, age for this query.

Example 1.3 ()

Therefore, given a query, there are certain indices that we can use or cannot use for it.

1. If we have a query `A >= 5`,
   (a) Can use hash index in general.
   (b) Can use tree index in general.
2. If we have query `A >= 5`,
   (a) Can use tree with index (A).
   (b) Can use tree with index (A, B).
   (c) Cannot use tree with index (B, A) since A is not prefix.
3. If we have query `A = 5`,
   (a) Can use hash with index (A).
   (b) Can use tree with index (A).
   (c) Cannot use hash with index (A, B) since hashing this tuple does not allow us to compare to A or retrieve it. It is one-way and pseudo-random.
   (d) Can use tree with index (A, B).
4. If we have `A = 5 AND B = 7`,
   (a) Can use hash with index (A, B).

Each index has its pros and cons, so why not just use both tree and hash indices? The problem is that when we modify a relation on the disk, we need to update the index as well. Therefore, having too many indices requires us to update more and takes more disk space.

Okay, so we can't use too many indices, but are indices *always* better than table scans? Not exactly.

Example 1.4 (Table Scans Wins)

Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$ with around 20% of R satisfying $A > v$ (could happen even for equality predicates). We need to follow pointers to get the actual result tuples.

1. IOs for scan-based selection is simply $B(R)$ (where we can retrieve multiple tuples in this block), while
2. IOs for index-based selection is the lookup-cost (to traverse down the tree) plus $0.2|R|$ (since for each tuple, we do a IO lookup, retrieve it, and then have to retrieve the next tuple which is likely not in the same block)

So table scan wins if a block contains more than 5 tuples since we might as well grab everything rather than look them up one by one.

Thankfully, the query optimizer will make these decisions for you.

1.5 Index Only Plans

Definition 1.14 (Index-Only Plans)

There are queries that can be answered only by accessing the index pages and not the data pages, known as **index-only plans**. For index-only plans, clustering is not important since we are looking only at the leaf nodes at most. Therefore, we only need to compute the I/O cost of traversing the tree and not to access data.

For equality, we can just compute the number of tuples in the index pages where the equality condition is satisfied. For ranges, we may need to traverse the leaf nodes, which will lead to additional I/O cost to retrieve the leaf index pages.

Example 1.5 (Index Only Queries)

If we look at the following query

$$\pi_A(\sigma_{A>v}(R)) \tag{4}$$

we see that we only care about the value of A and not the rest of the tuples, so we only need to look at the index pages and not the data itself.

Example 1.6 (Primary Index Clustered According to Search Key)

If we have a primary index, in most cases the actual records are also stored in the index pages/leaf nodes. If they are clustered according to attribute A, then one lookup can lead to all result tuples in their entirety. You can just hit a leaf and grab your records as you walk along the leaves.

Example 1.7 (Other Index-Only Queries)

For example, if we just wanted to look at the count of users with age 50, then we don't need the data. We can just look at the number of values in the leaf nodes of the B+ tree with this value.

```
SELECT E.do COUNT(*)
FROM Emp E
GROUP BY E.dno;
```

If we have an index on `(E.dno, E.sal)`, then the two queries are also index-only plans. However, if we index on `(E.dno)`, then we need to retrieve `E.sal` on the data page, incurring more cost.

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno;
```

```
SELECT AVG(E.sal)
FROM Emp E
GROUP BY E.dno;
```

Example 1.8 (Halloween Problem)

The Halloween problem refers to a phenomenon in databases where an update operation causes a change in the physical location of a row, potentially allowing the row to be visited again later in the same update operation. Look at the update.

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

This caused everyone to have a salary of 25000+. This is because when we updated someone with salary of say 1000, it went to 1100 and is moved further right in the B+ tree. Therefore, this is revisited again is increased again in the same update. To fix this, we could update the values in reverse, from the rightmost leaf node to the leftmost one so that increasing values are visited once. Or we can just create a to-do/done list that keeps track of which ones have been updated.

1.6 Exercises

Now let's go through a bunch of questions to clarify some of the IO cost computation.

Example 1.9 ()

Assume that we have the query `SELECT * FROM User WHERE age = 50;` with the following assumptions:

1. Assume 12 Users with `age = 50`.
2. Assume one data page (block) can hold 4 User tuples (so $f = 5$).
3. Suppose searching for data entry requires 3 IOs in a B+ tree, which contain pointers to the data records (so $h = 3$).

If the index is clustered, then we can just traverse down the B+ tree to get to the leaf node containing the value 50.

1. We have a cost of 3 to traverse down the tree to access the index pages which show the memory addresses of the tuples on disk.
2. Then, we will find entries and we want to find the cost to access the data pages. There are 12 Users, and for every address, we load the entire block in memory, which will retrieve the other users of age 50 (since this is clustered). At best, we will retrieve 3 blocks (of 4 tuples each) and at worst, due to block overlap, we will retrieve 4 blocks and read the rest from memory. This gives us a cost of +3 or +4.

The total cost is 6. If the index is unclustered, then we are not guaranteed that the data with values 50 will be contiguous, so we will in the worst case have to look at 12 different blocks, leading to a total cost of $3 + 12 = 15$.

Example 1.10 (Index Problem)

Consider a table `Orders(OrderID, CustomerID, OrderDate, TotalAmount)` with 5,000,000 records stored in 100,000 disk blocks (=pages or units of I/O). The rows are not sorted on any particular attribute. There is a **clustered** B+ tree index on OrderID (the primary key), and an **unclustered** B+ tree index on CustomerID.

Assume the following:

- Each node in the B+ trees corresponds to one disk block.
- The OrderID B+ tree has 4 levels (including the root) and 10,000 leaf nodes.
- The CustomerID B+ tree has 5 levels (including the root) and 50,000 leaf nodes.
- You do not have enough space in memory to hold all data pages.
- The root nodes of both B+ trees are always kept in memory.
- The non-root index nodes and all data pages are initially on disk.
- All the leaves have pointers to the next left node both on the left and the right side.
- All data pages are fully utilized.
- All nodes in B+ trees are also fully utilized.
- Uniformity in all places. You can ignore page boundaries.

Question 1.1

How many disk I/Os are required (index and data) to retrieve all order records for a specific CustomerID using the index on the CustomerID? Assume there are 100 orders for this customer.

We must access 4 levels, with 5 levels in tree minus 1 for head already in memory (+4). There is only 1 leaf index pages containing the matching entries, since there are 5m records and 50k leaf nodes, meaning that each leaf node can contain 100 addresses at most when it is a dense index. Given that there are 100 matches, we just need to use this leaf node to access the disk. This is not clustered, so we must look through all 100 addresses, reading each block from disk for an addition cost of 100 IOs. Total is **104 IOs**.

Question 1.2

Suppose the Orders table is frequently queried for recent orders, and the performance is critical. The current clustered index on OrderID is not providing optimal performance for these queries. You are considering reorganizing the Orders table to cluster it on OrderDate instead.

Assume that orders are uniformly distributed over time. Everything else is the same as in the original

question description **(i.e., the number of levels of the new B+ tree indexed on OrderDate is still 4 and the root is still in memory. And the number of leaf nodes is still 10,000)**. Ignore page boundaries.
Consider the following query:

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '2023-09-01' AND '2023-09-30';
```

How many disk I/Os are required in the worst case for the following query before and after the change? What is the impact of the change? Assume that 0.5% of the orders were made in September 2023.
For before, since the data is distributed uniformly, we must in the worst case check all data pages of the relation on disk.

1. We first have 3 IOs to go down to the leaf (4 levels minus 1 head already in memory).
2. We scan through all 10,000 leaves, but we are already on leaf 1, so need to traverse 9999 pointers to load each index leaf.

Therefore, the total IO for traversal is **10,002 IOs**. If we include the IOs for loading data pages, we also incur more costs. There are 100,000 disk blocks, so as we load all of them we incur an additional IO cost of 100,000, giving us **110,002 IOs**.
For after, if we cluster on OrderDate, then out of the 100,000 disk blocks, we assume that 0.5% of them, or 500 of them, will contain the relevant dates. Additionally, we assume that by uniformity, 0.5% of the 10,000 leaf nodes will contain these relevant addresses.

1. We traverse down to the leaf containing the address of the tuple with date attribute `2023-09-01`. This is 3 IOs.
2. We must traverse through the rest of the leaf nodes. Since we are already at the first one, we must load in an additional 49 leaf blocks, so 49 IOs.
3. As we load in each block, we are loading in a total of 500 data blocks from disk, so an additional 500 IOs.

We end up with **552 IOs**. If we exclude the IOs for loading the data pages, we incur fewer cost of **52 IOs** since this is only for traversal of the B+ tree.
The difference in these costs is just the first value minus the second value. For example, I can do

$$10,002 - 552 = 9450 \tag{5}$$

reduced IO costs.

Question 1.3

(a) How many disk I/Os are required in the worst case to insert a new order with order ID 2000 and customer ID 200 if updating the clustered index on the Order ID? (i.e., what is the I/O required for updating the index and inserting data records?)
(b) What would the result be if we updated the unclustered index on the CustomerID?

Assume all memory blocks are being used for this update only. Then we do not throw away an index page once it is read.
For (a),

1. You should first get 3 IOs to traverse down to the leaf.
2. Then you load the relevant data page from disk onto memory (1 IO).
3. Since this block is full and you want to add a tuple to it, you must split it into two pages. So you initialize two buffers in memory and write two separate blocks. Then you must write these blocks back into disk (2 IOs).
4. Then you split the leaf (assuming it's already in memory) by writing its 2 splits back, again by splitting it in memory (2 IOs). You this for the next node (since in worst case it's full), for a total of 3 more times as you traverse up the tree (6 IOs).
5. You finally want to write a new root into the disk, so you make one in memory and write it (1 IO).

This gives a total of **15 IOs**.
For (b), should be 17 or 18?

1. You should first traverse down to the leaf (+4 IOs).
2. Then you take a data page and write it in memory (+1 IO) since you don't need to split it to keep the clustering. Then you flush the new page to disk (+2 IOs).
3. Then you split the leaf node by writing its 2 splits back (+2). You do this 4 more times (+8 IOs).
4. You then want to write a new root node (+1). So in total +16 IOs.

2 Query Processing and Optimization

Definition 2.1 (Disk/Memory Blocks)

To set up some notation, let $B(R)$ represent the number of disk blocks that relation R takes up, and say M is the number of blocks available in our memory. We start off with some simple operations.

Example 2.1 (Basic Block Storage)

Suppose we have relation R with $|R| = 1000$ and each block can hold 30 tuples. Then $B(R) = \lceil 1000/30 \rceil = 34$, or 35 if there is overlap.

Definition 2.2 (Buffer Blocks)

It turns out that whenever we output data to stdout, the blocks need to be stored in a **buffer block**. If the buffer block is full, then it is flushed to stdout.

Example 2.2 (Cost of Querying Everything)

If we do `SELECT * FROM R`, then

1. we are at most retrieving $B(R)$ pages from disk to memory, so our IO cost is $B(R)$.
2. Our memory cost is 2 pages since we take each page, load it to memory, and put the answer in the output page. The next page (if any) can overwrite the previously loaded page.

We can stop early if we lookup by key. Remember that this is not counting the cost of writing the result out.

It turns out that the efficiency of most operations defined on modern DBMS depends on two things: sorting and hashing. We'll divide up our analysis this way, focusing on their applications in joining, which tends to be the mostly used and expensive operation. Note that there are many ways to process the same query. We can in the most basic sense just scan the entire relation in disk. We can sort it. We can use a tree or hash index, and so on. All have different performance characteristics and make different assumptions about the data, so the choice is really problem-dependent. What a DBMS does is implements all alternatives and let the *query optimizer* choose at runtime. We'll talk about the algorithms now and talk about the query optimizer later.

2.1 Brute-Force Algorithms

We start off with the most brute-force algorithms of theta joins. In here, we describe how to implement it, its IO cost, and its memory cost. In the following, when we compute $R \bowtie_p S$, R is called the **outer table** and S the **inner table**. Furthermore, another convention is that when we calculate IO cost, we *do not* factor in the cost of writing our result back to disk.

2.1.1 Nested Loop Joins

Our first algorithm simply takes in every block from R, and for every tuple $r \in R$, we take in every tuple in S to calculate the predicate $p(r, s)$.

Algorithm 2.1 (Nested Loop Join)

Nested-loop join just uses a brute-force nested loop when computing $R \bowtie_p S$.

Algorithm 1 Nested loop Join

```
Require: Outer table R, Inner table S
  function NestedLoop(R, S)
      for each block B_R ∈ R do
          load B_R into memory block M_1
          for each tuple r ∈ B_R do
              for each block B_S ∈ S do
                  load B_S into memory block M_2
                  for each tuple s ∈ B_S do
                      if p(r, s) is true then
                          Write r ⋈ s into buffer block M_3
                          Flush M_3 to stdout when it's full.
                      end if
                  end for
              end for
          end for
      end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R
2. For every tuple $r \in R$, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + |R| \cdot B(S) \tag{6}$$

with memory cost 3 (since we use M_1, M_2, M_3 blocks).

This is clearly not efficient since it requires a lot of IOs. We can make a slight improvement by trying to do as much as we can with the loaded blocks in memory.

Algorithm 2.2 (Block-Based Nested-Loop Join)

Block-based nested-loop join loops over the blocks rather than the tuples in the outer loops.

```
Require: Outer table R, Inner table S
  function BlockNestedLoopJoin(R, S)
      for each block B_R ∈ R do
          load B_R into memory block M_1
          for each block B_S ∈ S do
              load B_S into memory block M_2
              for each r ∈ B_R, s ∈ B_S do
                  if p(r, s) is true then
                      Write r ⋈ s into buffer block M_3
                      Flush M_3 to stdout when full.
                  end if
              end for
          end for
      end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R.

2. For every block B_R, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + B(R) \cdot B(S) \tag{7}$$

The memory cost is 3 (since we use M_1, M_2, M_3 blocks).

Algorithm 2.3 (Saturated Block-Based Nested-Loop Join)

The next optimization is to use more memory by basically stuffing the memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory.

Require: Outer table R, Inner table S
  function SatBlockNestedLoopJoin(R, S)
    for each set of blocks $\mathbf{B} = \{B_{i+1}, \dots, B_{i+(M-2)}\} \subset R$ **do**
      load $\mathbf{B}$ into memory blocks $M_1, \dots, M_{M-2}$.
      for each $B_S \in S$ **do**
        load B_S into memory block M_{M-1}.
        for each $r \in \mathbf{B}, s \in B_S$ **do**
          if $p(r, s)$ is true **then**
            Write $r \bowtie s$ into buffer block M_M.
            Flush M_M to stdout when it's full.
          end if
        end for
      end for
    end for
  end function

The total IO cost of this is

1. $B(R)$ to load R.
2. For every set of $M-2$ blocks, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S) \approx B(R) \cdot B(S)/M \tag{8}$$

You want to pick the bigger table as R since you want the smaller table S to be loaded/streamed in multiple times.

Algorithm 2.4 (Index Nested Loop Join)

If we want to compute $R \bowtie_{R.A=S.B} S$, the idea is to use the value of $R.A$ to probe the index on $S(B)$. That is, for each block of R, we load it into memory, and for each r in the block, we use the index on $S(B)$ to retrieve s with $s.B = r.A$, and output rs.

The IO runtime, assuming that S is unclustered and secondary, is

$$B(R) + |R| \cdot (\text{indexlookup}) \tag{9}$$

Since typically the cost of an index lookup is 2-4 IOs, it beats other join methods mentioned later if $|R|$ is not too big. Since this does not scale at all with S, it is better to pick R to be the smaller relation. The memory requirement as with other operations is 3 blocks.

2.2 Sort-Based Algorithms

2.2.1 External Merge Sort

Now let's talk about processing of queries, namely how sorting works in a database system, called **external merge sort**. In an algorithm course, we know that the runtime is $O(n \log n)$, but this is for CPU comparisons where the entire list is loaded in memory. This is extremely trivial in comparison to the IO commands we use in databases, so we will compute the runtime of sorting a relation by an attribute in terms of IO executions.

The problem is that we want to sort R but R does not fit in memory. We divide this algorithm into **passes** which deals with intermediate sequences of sorted blocks called **runs**.

1. Pass 0: Read M blocks of R at a time, sort them in memory, and write out the sorted blocks, which are called *level-0 runs*.
2. Pass 1: Read $M-1$ blocks of the level 0 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-1 runs*.[2]
3. Pass 2: Read $M-1$ blocks of the level 1 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-2 runs*.
4. ...
5. Final pass produces one sorted run.

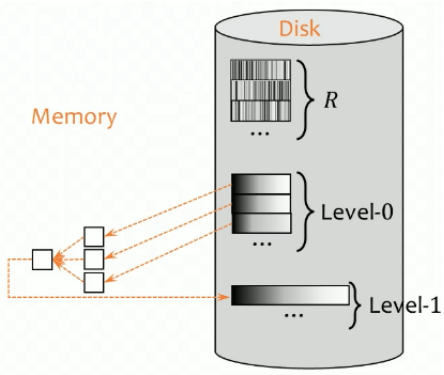


Figure 21

Algorithm 2.5 (External Merge Sort)

The implementation has a lot of details. Not finished yet.

[2]The reason we need $M-1$ rather than M is that now we are merging. We are not merging in-place so we need this extra buffer to store as we traverse our pointers down each of the $M-1$ blocks.

Require: Relation R
  function ExternalMergeSort(R)
    $L = [0]$ ▷ Array storing number of level-i runs
    while while there are blocks to read from R **do** ▷ First pass
      read the next M blocks $B = \{B_1, \dots, B_M\}$ at a time and store it in memory.
      sort B to generate a level-0 run $B^{(0)} = \{B'_1, \dots, B'_M\}$.
      write $B^{(0)}$ to disk.
      $L[0] += 1$
    end while
    while $L[-1] \geq M$ **do**
      append 0 to L
      let $\mathbf{B} = \{B\}$ be the set of previous runs
      while there exists blocks to be read in $\mathbf{B}$ **do**
        read $M-1$ blocks starting from the beginning of each run into memory.
        sort them to produce the ith run $B^{(i)}$.
        write $B^{(i)}$ to disk.
      end while
    end while
  end function

To compute the cost, we know that

1. in pass 0, we read M blocks of R at a time, sort them, and write out a level 0 run, so there are
$$\lceil B(R)/M \rceil \tag{10}$$
level 0 sorted runs, or passes.
2. in pass i, we merge $M-1$ level $(i-1)$ runs at a time, and write out a level i run. We have $M-1$ memory blocks for input and 1 to buffer output, so
$$\text{Num. of level i runs} = \left\lceil \frac{\text{Num. of level (i-1) runs}}{M-1} \right\rceil \tag{11}$$
3. The final pass produces 1 sorted run.

Therefore, the number of passes is approximately

$$\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1 \tag{12}$$

and the number of IOs is $2B(R)$ since each pass reads the entire relation once and write it once. The memory requirement is M (as much as possible).

Example 2.3 (Baby Merge)

Assume $M = 3$, with each block able to hold at most 1 number. Assume that we have an input (relation)

$$1, 7, 4, 5, 2, 8, 3, 6, 9 \tag{13}$$

Then we go through multiple passes.

1. Pass 0 will consist of 3 runs. You load each of the 3 numbers in memory and sort them.
$$1, 7, 4 \mapsto 1, 4, 7 \tag{14}$$
$$5, 2, 8 \mapsto 2, 5, 8 \tag{15}$$
$$9, 6, 3 \mapsto 3, 6, 9 \tag{16}$$

2. Pass 1. You merge them together by first taking 1 and 2, loading them in memory, and then comparing which one should go first. Once 1 is outputted, then the next number 4 overwrites 1 in memory, and then 2 is outputted, and so on.

$$1, 4, 7 + 2, 5, 8 \mapsto 1, 2, 4, 5, 7, 8 \tag{17}$$
$$3, 6, 9 \tag{18}$$

3. Pass 2. Merges the final two relations.

$$1, 2, 3, 4, 5, 7, 8 + 3, 6, 9 \mapsto 1, 2, 3, 4, 5, 6, 7, 8, 9 \tag{19}$$

Therefore, pass 0 uses all M pages to sort, and after that, when we merge, we only use $M - 1$ pages to merge the inputs together and 1 page for the output.

Some performance improvements include:

1. *Double Buffering*. You allocate an additional block for each run, and while you are processing (merging the relations in memory), you run the IO concurrently and store it in the new block to save some time.
2. *Blocked IO*. Instead of reading/writing one disk block at a time, we can read/write a bunch of them in clusters. This is sort of like parallelization where you don't output just one block, but multiple blocks done from multiple processing.

The problem with both of these is that we have smaller fan-in, i.e. more passes. Since we are using more blocks per run than we have, we can look at fewer runs at once.

2.2.2 Sort Merge Joins

Now that we know how to sort, let's exploit this to optimize joins beyond nested-loops. We introduce a naive version of sort-merge join.

Algorithm 2.6 (Naive Sort-Merge Join)

A clever way is to first sort R and S by their join attributes, and then merge. Given that the first tuples in sorted R, S is r, s, we do repeat until one of the R or S is exhausted.

1. If $r.A > s.B$, then $s =$ next tuple in S
2. Else if $r.A < s.B$, then $r =$ next tuple in R.
3. Else output all matching tuples and $r, s =$ next in R, S, which is basically a nested loop.

Therefore, given that it takes $\text{sort}(R), \text{sort}(S)$ to sort R, S (the equations above is too cluttered to write), the IO cost consists of

1. Sorting R and S.
2. We then must write R and S to disk in order to prepare for merging, so $B(R) + B(S)$.
3. We then must write R and S back into memory, one at a time, to merge them, so $B(R) + B(S)$.

Therefore, the IO is really just $2B(R) + 2B(S)$ more than it takes to sort both R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) + 2B(R) + 2B(S) \tag{20}$$

which is worst case $B(R) \cdot B(S)$ when everything joins.

Algorithm 2

Require:

function FUNC(x)
end function

Example 2.4 (Worst Case)

To see the worst case when the IO cost is $B(R) \cdot B(S)$, look at the following example. By the time we got to the first 3, we can't just increment the pointers for both relations. We must scan through all of the tuples of A with value 3 and all those in B with value 3 and do a nested loop to join them.

R:	S:
$r_1.A = 1$	$s_1.B = 1$
$\underline{r_2.A = 3}$	$s_2.B = 2$
$\underline{r_3.A = 3}$	$\underline{s_3.B = 3}$
$r_4.A = 5$	$\underline{s_4.B = 3}$
$r_5.A = 7$	$s_5.B = 8$
$r_6.A = 7$	
$r_7.A = 8$	

Figure 22: Before you increment the next pointer, you must loop through all combinations of the underlined elements in the left and right relations.

We have completely isolated the sorting phase and the merging phase, but we don't have to do this. Just like regular merge-sort, we can integrate them together to save IO in the last merge phase.

Algorithm 2.7 (Optimized Sort-Merge Join)

The algorithm is just slightly modified from the naive implementation. After the final pass from sorting both R and S, say that we have W_R and W_S final runs such that

$$M > W_R + W_S \tag{21}$$

We can assume that this is true since if it wasn't, we can just add another pass of external sort to reduce one of the W's. Then, we can do these 3 things simultaneously.

1. We can load in the next smallest block from R from its runs, merge them together.
2. We can load in the next smallest block from S form its runs, merge them together.
3. We can join the merged blocks *in memory* and output to the buffer to be flushed.

This saves us the IO cost of writing the sorted runs back into memory and then loading them again to write, giving us a total IO cost that is equal to that of simply sorting R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) \tag{22}$$

The memory varies depending on how many passes, but if R and S are moderately big in that we need full memory to sort them, then the memory cost is M (we use everything).

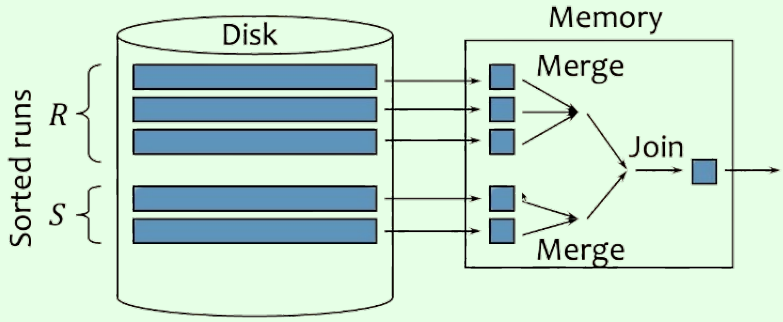


Figure 23: Sorting: produce sorted runs for R and S such that there are fewer than M of them total. Then in merge and join, we merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!

Example 2.5 (Two-Pass SMJ)

If SMJ completes in two passes, then the IOs is really cheap since we are basically getting a $2B(R) + 2B(S)$ cost of a level-0 pass, plus the final merge-join step which takes another $B(R) + B(S)$.

$$\text{IO} = 3(B(R) + B(S)) \tag{23}$$

If SMJ cannot complete in 2 passes, then we repeatedly merge to reduce the number of runs as necessary before the final merge and join.

2.2.3 Zig-Zag Join

Definition 2.3 (Zig Zag Join using Ordered Indices)

To compute $R \bowtie_{R.A=S.B} S$, the idea is to use the ordering provided by the indices on $R(A)$ and $S(B)$ to eliminate the sorting step of merge-join. The idea is similar to sort-merge join. We start at the leftmost leaf node of both indices of R and S, and traverse (right) through the leaves, querying both of the data at leaf a in R and b in S if the leaf values are equal.
Note that we don't even have to traverse through all leaves. If we find that a key is large, we can just start from the root node to traverse, possibly skipping many keys that don't match and not incurring all those IO costs. This can be helpful if the matching keys are distributed sparsely.

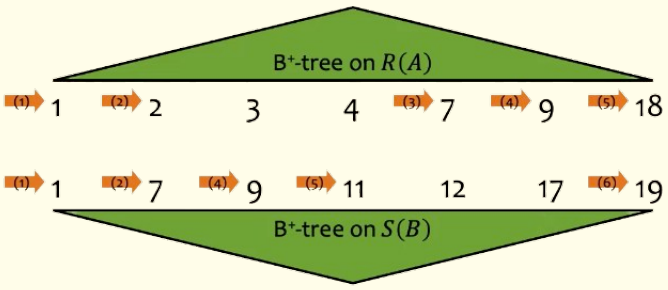


Figure 24: We see that the B+ tree of B has value 7 while that of A has value 2. Rather than traversing $2 \mapsto 3 \mapsto 4 \mapsto 7$, we can just traverse to 7 from the root node of A. This can give us a stronger bound.

2.2.4 Other Sort Based Algorithms

The set union, intersection, difference is pretty much just like SMJ.

For duplicate elimination, you simply modify it so that during both the sort and merge steps, you eliminate duplicates if you find any.

For grouping and aggregation, you do external merge sort by the group-by columns. The trick is you produce "partial" aggregate values in each run and combine them using merge.

2.3 Hash-Based Algorithms

2.3.1 Hash Join

Hash joining is useful when dealing with equality predicates: $R \bowtie_{R.A=S.B} S$.

Definition 2.4 (Hash Join)

The main idea of **hash join** is that we want to partition R and S by hashing (using a hash function that maps to $M-1$ values) their join attributes and then consider corresponding partitions (that get mapped to the same hash value) of R and S. If $r.A$ and $s.B$ get hashed to the same number, they might join, and if they don't, then they definitely won't join. The figure below nicely visualizes this.

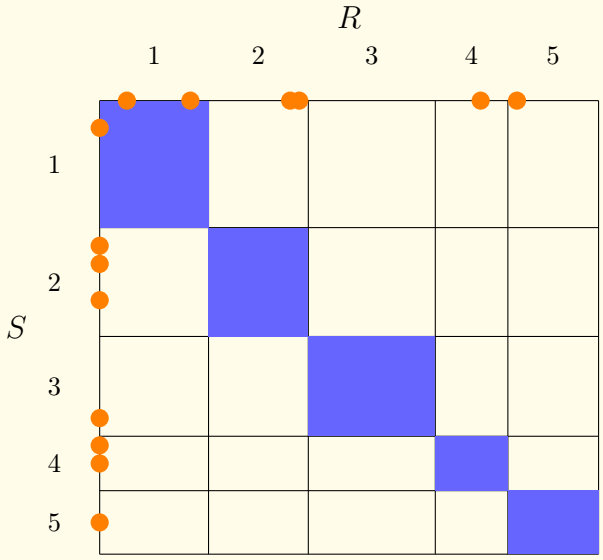


Figure 25: Say that the orange points represent the hashed values of $r.A$ and $s.B$ for $r, s \in R, S$. The nested loop considers all slots (all pairs of orange points between R and S), but hash join considers only those along the diagonal.

Then, in the **probing phase**, we simply read in each partition (the set of tuples that map to the same hash) of R into memory, stream in the corresponding partition of S, compare their *values* (not hashes since they are equal), and join if they are equal. If we cannot fit in a partition into memory, we just take a second hash function and hash it again to divide it into even smaller partitions (parallels merge-sort join).

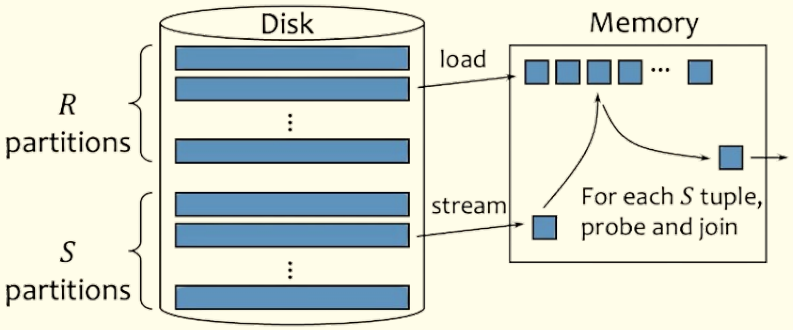


Figure 26

Therefore, if the hash join completes in two passes, the IO runtime is

$$3(B(R) + B(S)) \tag{24}$$

which is similar to merge-sort join. As for the memory requirement, let's first assume that in the probing phase, we should have enough memory to fit one partition of R, i.e. $M-1 > \lceil B(R)/(M-1) \rceil$, so solving for it roughly gives $M > \sqrt{B(R)} + 1$. We can always pick R to be the smaller relation, so roughly

$$M > \sqrt{\min\{B(R), B(S)\}} + 1 \tag{25}$$

Theorem 2.1 (Hash vs Sort-Merge Join)

To compare hash join and SMU, note that their IOs are the same, but for memory requirements, hash join is lower, especially when the two relations have very different sizes.

$$\sqrt{\min\{B(R), B(S)\}} + 1 < \sqrt{B(R) + B(S)} \tag{26}$$

Some other factors include the quality of the hash (may not generate evenly sized partitions). Furthermore, hash join does not support inequality joins unlike SMJ, and SMJ wins if either R and/or S is already sorted. SMJ also wins if the result needs to be in sorted order.

Sometimes, even block nested loop join may win in the following cases.

1. if many tuples join (as in the size of the join is $|S| \cdot |R|$) since we are doing unnecessary processing in hash/merge-sort joins.
2. if we have black-box predicates where we may not know the truth/false values of the θ

2.3.2 Other Hash Based Algorithms

The union, difference, and intersection are more or less like hash join.

For duplicate elimination, we can check for duplicates within each partition/bucket.

For grouping/aggregation, we can apply the hash functions to the group-by columns. Tuples in the same group must end up in the same partition/bucket. Or we may keep a running aggregate value for each group.

To compare the duality of sort and hash, note that

1. in sorting, we have a physical division and logical combination
2. in hashing, we have a logical division and physical combination

When handling large inputs,

1. in sorting we have multi-level merge
2. in hashing we have recursive partitioning

For IO patterns,

1. in sorting we have sequential write and random read (merge)
2. in hashing we have random write and sequential read (partition)

2.4 Exercises

Here are some exercises for calculating IO costs of these joins.

Example 2.6 (Join Operations)

Consider the two tables
1. `Orders(OrderID, CustomerID, OrderDate, TotalAmount)`
2. `Customers(CustomerID, Address)`

`Orders.CustomerID` is a foreign key referring to `Customers.CustomerID`.
The rows are not sorted on any particular attribute. We want to join the two tables using the attribute CustomerID:

$$\text{Orders} \bowtie_{\text{Orders.CustomerID} = \text{Customers.CustomerID}} \text{Customers}$$

The inner and outer tables can be swapped to reduce I/O costs in the following questions.
Assume the following:
- The cost metric is the number of page/block I/Os unless otherwise noted.
- DO NOT count the I/O cost of writing out the final result unless otherwise noted.
- M = 19 blocks (= pages) available in memory unless otherwise noted.
- Table Orders contains 50,000 records (rows/tuples) on disk. One block can contain 20 Orders-tuples.
- Table Customers contains 20,000 records (rows/tuples) on disk. One block can contain 10 Customers-tuples.
- Assume uniform distribution for Orders.CustomerID and Customers.CustomerID – i.e. the same number of Orders-tuples join with a Customers-tuple.
- Ignore page boundary wherever applicable.
- Assume uniform distributions wherever applicable.

Question 2.1
(a) For the Orders and Customers tables, we want to perform a nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost? (Choose the table that will reduce the I/O cost as the outer table.)
(b) For the Orders and Customers tables, we want to perform a block-based nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost?

For (a), we choose $R =$ Customers as the outer table since it is smaller. Therefore, our IO cost is

$$B(R) + |R| \cdot B(S) = 2000 + 20,000 \cdot 2500 = 50,002,000 \tag{27}$$

For (b), we also choose $R =$ Customers as outer since it's smaller. The IO cost is

$$B(R) + B(R) \cdot B(S) = 2000 + 2000 * 2500 = 5,002,000 \tag{28}$$

Question 2.2
(a) If we want to perform an external merge sort on the Orders table, how many level-0 runs does the external merge sort produce for the Orders table (M=19)?
(b) Continuing with the question (a), how many passes in total does the external merge sort take (including the first sorting pass)? Show the calculations for each pass (including the number of runs and size of each run).

(c) What is the total I/O cost of the external merge sort for the Orders table? Remember, do not count the cost of final write.
(d) Do an improved sort-merge-join, i.e., do merge and join in the same pass when the total number of sorted runs from Orders and Customers will fit in memory including an output block. Compute the minimum total I/O cost of sort-merge-join of table Orders and Customers.

For (a), we read M blocks of R at a time, so we need to have $\lceil B(R)/M \rceil = \lceil 2500/19 \rceil =$ **132 passes**.
For (b), we saw that

1. The level 0 pass takes 132 runs.
2. The level 1 pass takes $\lceil 132/(19-1) \rceil = 8$ runs.
3. At this point we can just run once more since $8 < 18$, so our level 2 pass takes 1 run.

So we have **3 passes** with **141 runs**.
For (c), we have $B(R) = 2500$ and just compute the following.

1. For level 0, we read all blocks, and write them all out: $2B(R)$.
2. For level 1, it's the same thing since we again read all blocks and write them back to disk: $2B(R)$.
3. For level 2, we just read through and do not include the final write, so $B(R)$.

This is a total of $5B(R) = 5 \cdot 2500 =$ **12,500 IOs**.
For (d),

1. For the first pass, you load all blocks of both relations in memory for the first iteration of sort merge and then write them to disk, giving us $2(B(R) + B(S))$ IOs.
2. The second pass is the same, giving us $2(B(R) + B(S))$ IOs.
3. By the third pass, we complete the sort-merge join giving us $B(R) + B(S)$.

This is a total of $5(B(R) + B(S)) = 5 \cdot (2000 + 2500) =$ **22,500 IOs**.

Question 2.3 Assume uniform distribution for the hash function(s).
(a) For the Orders and Customers tables, we want to perform a multi-pass hash join. How many passes do we need (including partitioning phase and join phase)?
(b) What is the minimum I/O cost of joining Orders and Customers using a hash join?
(c) What is the minimum number of memory blocks required if we want to perform a 2-pass hash join? Note that M should be an integer.

For (a), we must partition both relations.

1. In the first partition pass, we
   (a) take Customers and have $\lceil 2000/(19-1) \rceil = 112 > 18$. Each partition contains 112 blocks, which is too big for our memory, so we must partition again.
   (b) take Orders and have $\lceil 2500/(19-1) \rceil = 139$. Each partition contains 139 blocks, which is too big for our memory, so we must partition again.
2. In the second partition pass, we
   (a) take Customers and have $\lceil 112/18 \rceil = 7 \leq 18$, so we are done since each partition (of a partition) of 7 blocks can fit in memory.
   (b) take Orders and have $\lceil 139/18 \rceil = 8 \leq 18$, so we are done since each partition (of a partition) of 8 blocks can fit in memory.
3. Finally, we load each partition of Customers in memory (7 blocks), and for each partition, we just iterate through the corresponding partition of Customers. This is one join pass.

There are a total of **3 passes**, or 5 passes if we consider each partition step of each relation as an individual pass.
For (b), we see that since `Orders.CustomerID` is a foreign key, we won't have a case where every row in `Orders` will trivially join with every row in `Customers`. We can compute the steps as such.

1. The 1st partition pass.
   (a) For Customers, it requires you to read all blocks (2000 IOs) and then write $112 \cdot 18$ blocks back onto disk (2016 IOs),[a] or we are just approximating, this is about 2000 IOs as well.
   (b) For Orders, it requires you to read all blocks (2500 IOs) and then write $139 \cdot 18$ blocks back onto disk (2502 IOs), or just 2500 IOs as an approximation. This is flushed out as well.

This gives us a total of $2000 + 2016 + 2500 + 2502 = 9018$ IOs or with the approximations 9000 IOs.

2. The 2nd partition pass.
   (a) For Customers, now you have 18 partitions with each 112 blocks. For each partition, you load it again (112 IOs) and then partition it again to write back $18 \cdot 7$ blocks (126 IOs) for a total of 238 IOs. You do this 18 times for each partition giving us $238 \cdot 18 = 4284$ IOs. Again, we can just approximate it by saying that loading all partitions is 2000 and writing all is 2000, giving us 4000 IOs.
   (b) For Orders, now you have 18 partitions with each 139 blocks. For each partition, you load it again (139 IOs) and then partition it again to write back $18 \cdot 8$ blocks (144 IOs) for a total of 283 IOs. You do this 18 times for each partition giving us $283 \cdot 18 = 5904$ IOs. Again, we can just approximate it by saying that loading all partitions is 2500 and writing all is 2500, giving us 5000 IOs.

   This gives us a total of $4284 + 5904 = 10188$ IOs, or with the approximations 9000 IOs.
3. In the join phase, you load the partitions of R and S with the matching hashes once each to compare, so we have (not including write)
   (a) the partitions of R, which is $18 \cdot 18 \cdot 7 = 2268$ IOs, or 2000 IOs approximately.
   (b) the partitions of S, which is $18 \cdot 18 \cdot 8 = 2592$ IOs, or 2500 IOs approximately.

   giving us a total of 4860 IOs.

Therefore, we have a total of $9018 + 10188 + 4860 =$ **24066 IOs**, or if we use our approximations, it's simply $5(B(R) + B(S)) = 5 \cdot 4500 =$ **22,500 IOs**.

For (c), M must satisfy

$$M \geq \lceil \sqrt{\min\{2500, 2000\}} \rceil + 1 = 46 \tag{29}$$

giving us $M = 46$. Checking this is indeed the case, since now the image of our hash function is $\{1, \dots, 45\}$, and assuming uniformity the number of blocks in each partition is $\lceil 2000/45 \rceil = 45$, which is just enough to fit in memory and then use the last block as an input buffer for the bigger relation when joining.

2.5 Logical Plans

When the DBMS chooses the best way to sort or merge two relations, it needs to choose it immediately. This can be done crudely by looking at the statistics of the relations that it is working with, but it doesn't guarantee that you will get the optimal plan. Therefore, it goes by the principal that you shouldn't try and waste time choosing the optimal one, but rather avoid the horrible ones. As a user of this DBMS, we should also take care in writing queries that are not too computationally or IO heavy. Two general heuristics that we should follow are:

1. You want to *push down*, i.e. use as early as possible, selections and projections.
2. You want to join smaller relations first and avoid using cross product, which can be devastating in memory.

Definition 2.5 (Logical Plan)

To have an approximate sense of how computationally heavy a query is, we can construct a high-level **logical plan**, which shows the computation DAG of the relational operators that we will perform for a query. We can optimize the logical plan by modifying our intermediate steps, such as optimizing our relational algebra logic or our implementation of SQL.

[a] This is not exactly 2000 because $2000/18 = 111.11$, which means that after repeatedly flushing out the filled partitions to disk 111 times, at the end we will have a partially filled buffer block in memory for each of the 18 partitions. This will need to be flushed out as well.

2.5.1 Query Rewrite

Using what we know, we can get a bit more theoretical and use the following identities in relational algebra. However, this has already been optimized and only does so much in practice.

Theorem 2.2 (Identities)

The following hold:

1. Selection-Join Conversion: $\sigma_p(R \times S) = R \bowtie_p S$
2. Selection Merge/Split: $\sigma_{p_1}(\sigma_{p_2} R) = \sigma_{p_1 \wedge p_2} R$
3. Projection Merge/Split: $\pi_{L_1}(\pi_{L_2} R) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
4. Selection Push Down/Pull Up: $\sigma_{p \wedge p_r \wedge p_s}(R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \wedge p'} (\sigma_{p_s} S)$, where:
   (a) p_r is a predicate involving only R columns
   (b) p_s is a predicate involving only S columns
   (c) p and p' are predicates involving both R and S columns
5. Projection Push Down: $\pi_L(\sigma_p R) = \pi_L(\sigma_p(\pi_{LL'} R))$, where L' is the set of columns referenced by p that are not in L

Definition 2.6 (SQL Query Rewrite)

We can rewrite SQL queries directly, though this is more complicated and requires knowledge of the nuances of the DBMS.

1. Subqueries and views may not be efficient, as they divide a query into nested blocks. Processing each block separately forces the DBMS to use join methods, which may not be optimal for the entire query though it may be optimal for each block.
2. Unnest queries convert subqueries/views to joins.

Therefore, it is usually easier to deal with select-project-join queries, where the rules of relational algebra can be cleanly applied.

Example 2.7 (Query Rewrite)

Given the query, we wish to rewrite it.

```
SELECT name
FROM User
WhERE uid = ANY(SELECT uid FROM Member);
```

The following is wrong since there may be one user in two groups, so it will be duplicated.[a]

```
SELECT name
FROM User, Member
WHERE User.uid = Member.uid;
```

The following is correct assuming `User.uid` is a key.

```
SELECT name
FROM (SELECT DISTINCT User.uid, name)
FROM User, Member
WHERE User.uid = Member.uid);
```

[a] A bit of review: when testing whether two queries are equal, think about if the two queries treat duplicates, null values, and empty relations in the same way.

Example 2.8 (Correlated Subqueries)

Look at this query where we want to select all group ids with name like Springfield and having less than some number of members.

```
SELECT gid
FROM Group, (SELECT gid, COUNT(*) AS cnt FROM Member GROUP BY gid) t
WHERE t.gid = Group.gid AND min_size > t.cnt
AND name LIKE 'Springfield%';
```

This is inefficient since for every `gid`, we are making an entire extra query to select the counts. This is called a **non-correlated** query since this subquery is being run independently for every run. It ends up computing the size of *every* group, unlike the following one, where it filters out groups named Springfield first and then computes their size.

```
SELECT gid FROM Group
WHERE name LIKE 'Springfield%'
AND min_size > (SELECT COUNT(*) FROM Member WHERE Member.gid = Group.gid);
```

2.5.2 Search Strategies

Given a set of operations we have to do, the number of permutations that we can apply these operations grows super-exponentially. The problem of finding the best permutation is called a **search strategy**.

Example 2.9 (Left-Deep Plans)

Say that we have relations $R_1, \dots, R_n$ that we want to join. The set of all sequences in which we can join them is bijective to the set of all binary trees with leaves R_i. This grows super-exponentially, reading $30,240$ for $n = 6$. There are too many logical plans to choose from, so we must reduce this search space. Here are some heuristics.

1. We consider only **left-deep** plans, in which case every time we join two relations, it is the outer relation in the next join.[a]

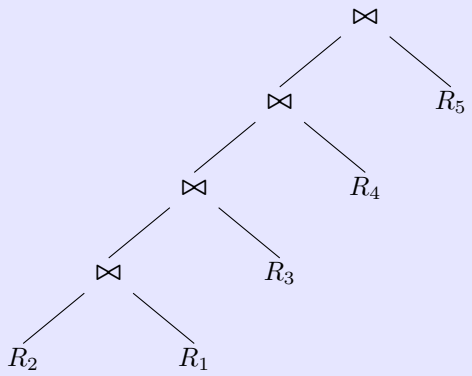


Figure 27: Left deep plans have a search space of only $n!$, which is better than before.

2. We can consider a balanced binary tree, which can be parallel processed, but this causes more runtime on the CPU in sort-merge joins, you must materialize the result in the disk, and finally the search space of binary trees may be larger than that of the left-deep tree.

[a] Note that since the right/inner relation is the one that is being scanned, we want the right one to be smaller since for each block of the left relation, we are looping over all blocks of the right relation. Therefore, left-deep plans are much more efficient

Even left-deep plans are still pretty bad, and so optimizing this requires a bit of DP (dynamic programming), using *Selinger's algorithm*.

Algorithm 2.8 (Selinger's Algorithm)

Given $R_1, \dots, R_n$, we must choose a permutation from the $n!$ permutations. Say that the cost of the optimal join of a set $\mathbb{R}$ is $f(\mathbb{R})$. Note the recursive formula for some $S \subset [n]$.

$$f(\{R_i\}_{i \in S}) = \min_i f(\{R_j\}_{j \in S, j \neq i}) + f(R_i, \bowtie_{j \in S, j \neq i} R_j) \tag{30}$$

Where we sum up the cost of getting the accumulated relation and add it to the additional cost of joining once more with R_i. Therefore, given the R_i's,

1. We compute all $f(\{R_i, R_j\})$ for $i < j$ (since $j > i$ requires us the larger one to be inner).
2. Then we apply the recursive formula for all 3-combinations and so on, until we get to n-combinations.

Given a certain logical plan, the DBMS tries to choose an optimal physical plan as we will see later. However, the globally optimal plan may not be achieved with a greedy approach of first choosing the optimal logical plan and then its optimal physical plan. Due to the sheer size of the search space, we tend to go for "good" plans rather than optimal ones.

2.6 Physical Plan

The logical plan gives us an abstract view of the operations that we need to perform. It is mainly defined at the relational algebra or language level. But simply sorting or joining two relations is not done in just one way (e.g. we can scan, hash, or sort in different ways). Optimizing the logical plan may or may not help in the runtime, since operations are dependent on the size of the intermediate inputs.

Definition 2.7 (Physical Plan)

The actual sub-decisions needed to execute these operations constitute the **physical plan**, which is the actual implementation including even more operations in between each node of the logical plan. Here are a few terms to know.

1. *On the fly.* The computations are done in memory and are not written back to disk.
2. *(Index) Scan.* We scan for index value using a clustered/unclustered index on a B+ tree.
3. *Sort.* Usually means external merge sort.
4. *Filter.* Means the same as selection.

The difference between the logical and physical plan is that the logical plan represents *what* needs to be done (which we write SQL) and not *how* (which the DBMS chooses). Consider the two approaches.

Example 2.10 (Query Plans)

Consider the following SQL query:

```
SELECT Group.title
FROM User
JOIN Member ON User.uid = Member.uid
JOIN Group ON Member.gid = Group.gid
WHERE User.name = 'Bart';
```

since we don't have to scan the huge relation from the disk multiple times. We can just send it directly to the next join.

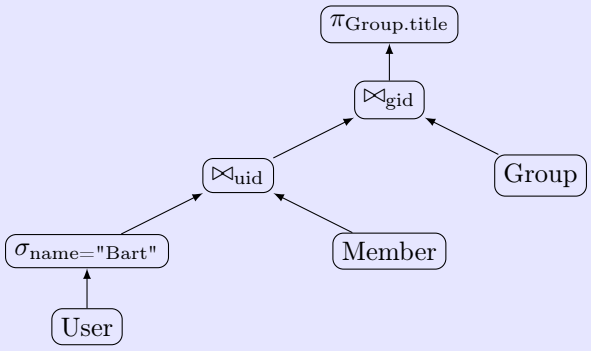


Figure 28: Logical Plan. We first take User, select Bart, and join it to Member over uid. Then we join it with Group on gid, and finally project the title attribute.

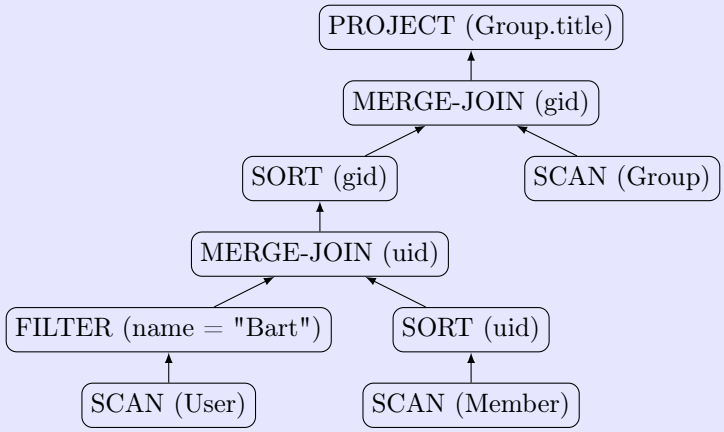


Figure 29: Physical Plan. We first take User and do a scan before filtering/selecting tuples with Bart. We also scan Member and sort it by uid in order to prepare for merge-join. Once we merge-join over uid, we sort it again to prepare a second merge-join with Group (which we scan first). Once we do this, we finally project the title attribute.

2.6.1 SQL Rewrite

At the language level, SQL provides some APIs to force the DBMS to use a certain physical plan if desired. This requires expertise and should not be done by beginners, however.

2.6.2 Cardinality Estimation

In the physical plan, we need to have a cost estimation for each operator. For example, we know that `SORT(gid)` takes $O(B(\text{input}) \cdot \log_M B(\text{input}))$, but we should find out what B, the number of blocks needed to store our input relation, is. To do this, we need the size of intermediate results through cardinality estimation.

Usually we cannot do quick and accurate cardinality estimation without strong assumptions, the first of which is uniformity of data.

Example 2.11 (Selection with Equality Predicates)

Suppose you have a relation R with $|R| = 100,000$ tuples. Assume that it has an attribute A taking integer values in $[50, 100)$ *distributed uniformly*. Then, there are 50 distinct values, and when we want

to do $\sigma_{A=a}(R)$, then we would expect it to return

$$|\sigma_{A=a}(R)| = \frac{|R|}{|\pi_A(R)|} = 2000 \tag{31}$$

tuples.

The second assumption is *independence* of the distributions over each attribute.

Example 2.12 (Selection with Conjunctive Predicates)

If we have the same relation R with integer attributes $A \in [50, 100), B \in [10, 20)$ independently and uniformly distributed. Then,

$$|\sigma_{A=a,B=b}(R) = \frac{|R|}{|\pi_A(R)| \cdot |\pi_B(R)|} = \frac{100,000}{50 \cdot 10} = 200 \tag{32}$$

At this point, we are just using inclusion-exclusion principle and this becomes a counting problem.

Example 2.13 (Negated, Disjunctive Predicates)

We list these identities for brevity. The math is pretty simple.

$$|\sigma_{A\neq a}(R)| = |R| \cdot \left(1 - \frac{1}{|\pi_A(R)|}\right) \tag{33}$$

and using I/E principle, we have

$$|\sigma_{A=a\vee B=b}(R)| = |R| \cdot \left(\frac{1}{|\pi_A(R)|} + \frac{1}{|\pi_B(R)|} - \frac{1}{|\pi_A(R)| \cdot |\pi_B(R)|}\right) \tag{34}$$

Example 2.14 (Range Predicates)

Range also works similarly, but only if we know the actual bounds of the attribute values.

$$|\sigma_{A>a}(R)| = |R| \cdot \frac{\max(R.A) - a}{\max(R.A) - \min(R.A)} \tag{35}$$

Clearly, if we know that an attribute follows, say a Gaussian or a Poisson distribution, we can just calculate the difference in the CDFs and scale up by the relation size to get the approximate cardinality. I think this is what the professor refers to as *histogram estimation*.

For joins, we need yet another assumption, called *containment of value sets*. This means that if we are natural joining $R(A, B) \bowtie S(A, C)$, every tuple in the smaller (as in fewer distinct values for the join attribute A) joins with some tuple in the other relation. In other words,

$$|\pi_A(R)| \leq |\pi_A(S)| \implies \pi_A(R) \subset \pi_A(S) \tag{36}$$

which again is a very strong assumption in general but holds in the case of foreign key joins.

Example 2.15 (Two Way Equi-Join)

With the containment assumption, we have

$$|R \bowtie_A S| = \frac{|R| \cdot |S|}{\max(|\pi_A(R)|, |\pi_A(S)|)} \tag{37}$$

Think of this as looking at the cross product between the two relations, and then filtering out the actual tuples that don't belong there.

2.7 Exercises

Let's do a more comprehensive exercise.

Example 2.16 (Cost Estimation of Physical Query Plan)

Say we have three relations

1. `Student(sid, name, age, addr)`. $T(S) = 10,000$ tuples, $B(S) = 1,000$ pages.
2. `Book(bid, title, author)`. $T(B) = 50,000$ tuples, $B(S) = 5,000$ pages.
3. `Checkout(sid, bid, date)`. $T(C) = 300,000$ tuples, $B(C) = 15,000$ pages.

And say that the number of `author` attribute values in `Book` with $7 \leq$ `age` ≤ 24 is 500 tuples. There is an unclustered B+ tree index on `B.author`, a clustered B+ tree index on `C.bid`, and all index pages are in memory. Also we assume unlimited memory to simplify things.

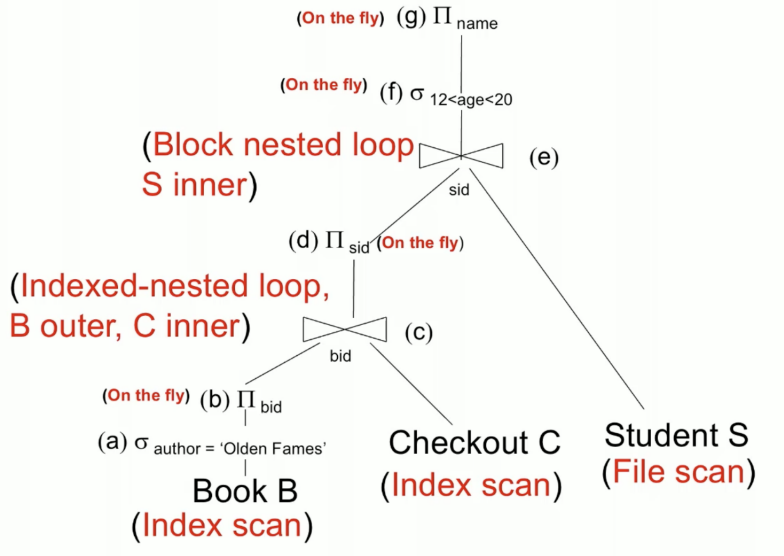


Figure 30: We have the following physical plan.

Okay, so let's do this. Note that since all index pages are in memory, we are storing the entire B+ tree in memory and don't need extra IOs to traverse it.

a) For the selection, the author is an unclustered B+ tree. There are 50,000 tuples and 500 distinct authors. Since we're querying 1 author, by uniformity we would need to access 100 book which may all be in their own disk page, so we need 100 IOs.[a] We end up with an output relation of 100 tuples in memory, so we also have a cardinality of 100.

$$\text{IO}(a) = \frac{T(B)}{500} = \frac{50,000}{500} = 100, \ \text{Card}(a) = 100 \tag{38}$$

b) For the projection on `bid`, we have already loaded in our relation in memory, so the IO cost is 0. We are still working with 100 pages, so our cardinality is still 100.

$$\text{IO} = 0, \ \text{Card}(b) = 100 \tag{39}$$

c) Now we do a join with an index-nested loop join on `bid`. Recall that we want to use the value of the outer table `R.bid` to probe the index on the inner table `C.bid`, which is clustered. We

already have our outer table in memory, and we use the index `bid` to probe our inner table `C`. For each of the 50,000 book tuples in `B`, there are 300,000 checkout tuples in `C`, meaning that there are about $300,000/50,000 = 6$ checkout per book. For each of the 100 book tuples in (a), we expect to get 6 checkouts per book. There are $300,000/15,000 = 20$ checkout tuples per page, so counting for page boundaries we assume that 6 tuples will fit in at most 2 pages (or maybe 1). Therefore, we have

$$\text{IO}(c) = 100 \cdot 2 = 200, \ \text{Card}(c) = 100 \cdot 6 = 600 \tag{40}$$

d) This is done in memory so IO is 0. Note that we have a total of 600 checkout/book tuples. Since this is a projection, the cardinality also remains the same.

$$\text{IO}(d) = 0, \ \text{Card}(d) = 600 \tag{41}$$

e) Now we have a block nested loop join. Since (d) is already in memory (on the fly), all we have to do is load all of `S` into memory (unlimited), which means our IO cost is $B(S) = 1000$. We are joining with the student relation, and since there is 1 student per checkout, our output relation is still 600 tuples long.

$$\text{IO}(e) = 1000, \ \text{Card}(e) = 600 \tag{42}$$

f) Finally we select, and assuming that the ages are uniformly distributed, we expect $(20 - 12 - 1)/(24 - 7 + 1) = 7/18$ of the relations to remain after selection. IO is 0 since this is on the fly.

$$\text{IO}(f) = 0, \ \text{Card}(f) = 600 \cdot \frac{7}{18} \approx 234 \tag{43}$$

g) Finally, we project onto name. IO is 0 since on the fly. We are projecting on names and assuming we don't remove duplicates[b] our output relation is still the same length.

$$\text{IO}(g) = 0, \ \text{Card}(g) = 234 \tag{44}$$

The total cost is $1000 + 200 + 100 = 1300$ and the final cardinality is 234.

[a] If this was clustered, then each page can store 10 tuples, so we actually need 10 IOs.
[b] Is this really a realistic assumption?