CS145: Study Guide

Abstract

These study notes are to make sure you understand the high-level concepts.

- **Please consult the slides first** as we've tried to make them a complete reference for the course.
- If you are confused, please ask questions on Piazza–we're happy to update these notes! (You'll notice below that there is a "Notes Compiled" time to keep track of versions.)
- These notes may contain more color about topics not discussed in lecture. Unlike lecture and homework, material covered in these notes is **NOT** necessarily required knowledge. However material that is not required knowledge will mostly be displayed in boxes which say *"Extra Material"*
- Finally, these are *VERY* rough notes, but we hope they help you!

Notes Compiled: Tuesday 1st November, 2016 at 23:27

Contents

1 Transactions

[Lectures 8-9]

A *transaction (TXN)* is a sequence of one or more operations (reads or writes), which reflect a single real-world transition (which as in the real world, should happen either completely or not at all). Grouping changes to the state of a database into TXNs allows:

- One or more users to run multiple queries concurrently, with a high level of performance, without having to take concurrency into account
- DBMS recovery from crashes or user-initiated aborts to a consistent state without any additional user supplied code.

1.1 The ACID Properties

- **A**tomicity: The DBMS state either reflects all effects of a transaction, or none of them.
- **C**onsistency: If (explicit and implicit) integrity constraints hold over the state before a transaction, they will also hold over the state after the transaction
- **I**solation: The effect of transactions concurrently is the same as the effect of running all transactions one-at-a-time
- **D**urability: Once a transaction has committed, its effects remain in the database (are stably stored on disk, flash, or other non-volatile storage)

ACID Challenges

- Power failures
- User-forced aborts/rollbacks
- Concurrency (DBMS can freely interleave individual actions of separate transactions)
- Performance

1.2 TXNs In SQL

- Defaults to one transaction per SQL statement
- START TRANSACTION to begin and COMMIT or ROLLBACK to end can manually specify a transaction

1.3 Logging for Atomicity & Durability

1.3.1 The Log

Goal: Ensure that operations (e.g. from partially-completed TXNs) can be undone, so that we can write partial TXNs to disk and still ensure atomicity

- A list of modifications: the basic idea is to record UNDO information for every update, writing a diff (minimal info) to the log sequentially
- Log records contain (XID, location, old data, new data), which is enough to undo any transaction (Here XID is the TXN id)
- Duplexed and archived on stable storage (we assume the log is not lost)
- Can *force writes* to disk.
- Handled internally by the DBMS (transparent to users)

1.3.2 Write-Ahead Logging

- Basic protocol:
    1. All log records for the TXN force-written to disk
    2. Commit record written to disk **& TXN commits**
    3. Data written to disk
- Must force log record to disk for an update before the corresponding data page goes to storage (guarantees Atomicity)
- Must write all log records for a transaction before commit. (guarantees Durability)
- The commit is guaranteed when the commmit record hits disk.

1.4 Concurrency with Isolation & Consistency

1.4.1 Scheduling Definitions

- A *schedule* is a particular interleaving of the operations of multiple TXNs
- *Serial Schedule:* A schedule that does not interleave the actions of different transactions
- *Equivalent Schedules:* Two schedules such that executing them have an identical effect on the database state, for any possible initial database state.
- *Serializable Schedule:* A schedule that is equivalent to *some* serial schedule for the transactions scheduled.

1.4.2 Anomalies

- Lost update/Overwriting uncommitted data (Write after Write)
- Dirty read (Read after Write; writing transaction subsequently aborts)
- Inconsistent read (A task sees some but not all changes made by another)

1.4.3 Conflicts

- Two actions *conflict* if they are:
  - Part of different TXNs
  - Involve the same variable
  - At least one is a write
- Note that a conflict is **not** an inherently bad occurrence, nor does it imply an anomaly- just a definition

1.4.4 Locking

Ensure that each transaction's view of the DB state is 'consistent.'

1.4.5 Two-Phase Locking (2PL) Protocol

- In 2PL, there are two phases of locking: lock acquisition and lock release. Once TXN releases a lock, it cannot acquire any more locks.
- Each transaction must obtain a *S (shared) lock* on an object before reading it, and a *X (exclusive) lock* on an object before writing it
  - An S lock can be held by multiple TXNs
  - An X lock can be held by only one TXN
  - An X lock cannot be held on an object at the same time as any other S or X locks on that object
- All locks held by a transaction are released when the transaction completes.
- We use Strict 2PL in this course, which has the added requirement over 2PL that a transaction releases locks only at the end of the transaction.

Strict 2PL guarantees that schedules are conflict serializable, but does not allow all serializable schedules. See lecture for diagrams comparing these two protocols.

1.4.6 Conflict Equivalence

We define conflict equivalence in order to capture the behavior of locking, keep that in mind and this may make more sense!

Two schedules are *conflict equivalent* if:

- The schedules involve the same actions of the same transactions
- Every pair of conflicting actions of the two TXNs are ordered in the same way

A schedule is *conflict serializable* if it is conflict equivalent to *some* serial schedule. Conflict serializable schedules are a subset of serializable schedules.

1.4.7 Dependency Graphs

A dependency graph consists of a directed graph with:

- One node per transaction $T_1 \dots T_N$

- An edge from T_i to T_j if they conflict on an object and the action of T_i precedes T_j.

With Strict 2PL, a schedule is *conflict serializable* if and only if its dependency graph is acyclic.[1] To see the forward direction, observe that a conflict serializable graph that uses Strict 2PL and does not deadlock ensures that there cannot be a cycle in the dependency graph: All locks are held until the end of the transaction. Hence, if there were a cycle this would cause a deadlock. On the other hand, if the dependence graph for a schedule S is acyclic, then one can use any topological order for the graph that agress with the edges to create a serial schedule. Notice that S is conflict equivalent to the serial schedule, since each conflicting actionis ordered in the same way. Hence, S is conflict equivalent to a serializable schedule, hence conflict serializable.

One possibly helpful way to remember this is: *strict 2PL (and locking in general) converts bad (cyclic) conflicts into deadlocks.* We describe deadlocks more next.

1.4.8 Deadlock

A **deadlock** is a cycle of transactions waiting for locks to be released by one another (and thus unable to proceed).

Example 1.1 *Here is a schedule that could deadlock:*

$$T_1 : W(A); \quad T_2 : W(B); \quad T_2 : R(A); \quad T_1 : R(B)$$

If we use locking, the deadlock occurs at the last transaction. Here, T_1 *holds the lock on* A*, which causes* T_2 *to block waiting for* T_1*. Before blocking,* T_2 *has acquired an exculsive lock on* B*.* T_1 *attempts to acquire this lock from* T_2*, but now it blocks. Neither can make progress, and they are deadlocked.*

[1] This is true more generally, but we did not consider those issues in this course.

Extra Material: Deadlock Prevention and Detection

There are two common ways to handle deadlock: Deadlock Prevention and Deadlock Detection. Detection is more common.

Deadlock Prevention Assign priorities to transactions based on timestamps. Lower timestamps have higher priority.

- If transaction T_i wants a lock held by T_j, then act on either a *wait-die* or *wound-wait* policy.
- Wait-die: If T_i has higher priority, T_i waits for T_j; otherwise T_i aborts.
- Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits.
- In either case, make sure that restarted transactions keep their old timestamp, so they eventually have the priority to succeed.

Deadlock Detection We create a *waits-for graph* and check this graph for cycles periodically.

- There is a single node for each transaction. We will abuse notation and conflate the node and the transaction it represents.
- There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock.
- If the graph has a cycle, then there may be deadlock[2]
- Periodically check for cycles. If a cycle is found, kill a transactions on the cycle to break the deadlock.
- The check for a cycle and deadlock could be expensive if there are large cycles. However, typically the cycles in the graph tend to be small, and one can optimize for this.

1.5 Some Highlights and Tips

Note there are examples in the Lectures 8-9. Check them out! Serializability is the database gold standard notion of correctness. It allows us to interleave the order of transactions for performance. Conflict serializability is a restricted notion that captures what our locking protocol allows.

- Intuitively, to see if a schedule is equivalent to a fixed serial schedule transactions- say $T_1, \ldots, T_k$- you simulate running these transactions. You check that for each object they write, the schedule and the serial schedule have the *same value at the end of both schedules.* It does not matter if the two databases have different values in the middle of execution!

- To test if a schedule is *serializable*, you can do the following. Suppose the schedule contains k transactions, $T_1, \ldots, T_k$. You consider whether that schedule is equivalent to *some* ordering of the transactions. You simply try the algorithm from the preceding bullet point on each possible orders (all $k!$ orders). This is definitely a tougher notion to test!

- If no such ordering exists, the transaction is *not* serializable. Intuitively, it means that there is some kind of dependence like in Lecture 12, where some examples illustrate this.

- To test if a schedule is *conflict serializable*, you simply run the strict 2PL locking protocol and see if you get stuck! More precisely, you form the conflict graph and check if it is acyclic.

- Not all serializable schedules are conflict serializable. This makes sense as conflict Serializability is a more strict notion than serializability!

- You could still have a deadlock even when running Strict 2PL. If you don't have a deadlock, and you run strict 2PL to completion for a schedule, then your schedule is serializable. This captures the sense in which locking is correct.

2 Storage

[Lecture 12]

We study hard disks, which we assume are durable (we do not worry about failure, which is handled by other mechanisms). Disks are block devices.[3] This means we can only read or write data at block sizes (4K, 8K or so). We need to bring the data into memory to modify it or have the CPU read it, a region of memory called the buffer. The buffer may be smaller than all available memory.

2.1 Disk Mechanics

These are not important for the exam.

- Disk storage hardware is comprised of multiple *platters*.
- A *track* is a ring in a platter (bytes per track on the order of 10^5)
- A *cylinder* is one track from every platter (the track in the same location on each platter)
- Read and write from the disk in units of disk blocks (typically 4K, 8K, 16K)

2.2 Disk Latency

These are not important for the exam.

- Disk Latency: Total time from when command is issued to when data is in memory
- Comprised of *seek time* + *rotational latency*
- Sequential disk access much more efficient than random disk access
- Seek time comes from the time needed for the disk head to reach the track on which the block to be read lies

[3] We will conflate blocks and pages for this course.

2.3 High-level: Disk vs. Main Memory

- Accessing (reading from / writing to) the disk is slow, whereas accessing main memory is fast; in particular:
  - Main memory is $\approx$ 10x faster for *sequential access*
  - Main memory is $\approx 100,000$x faster for *random access* (i.e. not sequential)
- Disk is *durable* (we assume data is safe once on disk for this course) while main memory is *volatile*- data can be lost if crash occurs, etc.
- Main memory is far more expensive per unit of storage space than disk- we'll assume in this course that we have (effectively) unlimited disk space but limited main memory

2.4 Cost Model

We will count cost in what is called the *IO cost model* throughout the rest of these notes: We count the number of pages read and written to disk and ignore CPU costs. These costs are a simplification of the true costs sufficient to capture the rough trends. The message of these costs models is that it is indeed possible for the machine to estimate your cost; there are some hard elements like statistics, but we exposed you to them so you would know how to write efficient queries.

Remark 2.1 *The buffer is not all the memory of the machine (there are registers, a small amount of scratch space, etc.). It is however the place where all* data pages *are accessed. We count only access to the data pages themselves in our model. We will also not care too much about the mechnanics of the buffer pool, but its size can be important for performance.*

3 Relational Algebra

There is a formal reference for all of the relational model (Chapter 3) of Abiteboul, Hull and Vianu available for free from `http://webdam.inria.fr/Alice/`. We give an informal treatment here.

Relational Model Fix a set of attributes $\mathcal{A}$. Given an attribute $A \in \mathcal{A}$ let $D(A)$ be the corresponding domain for that attribute. For example, if A is age, then $D(A)$ could be the set of non-negative integers.

A **schema** for a relation is a symbol name and a set of attributes written $R(\bar{A})$ in which R is the symbol name for the relation, and $\bar{A} \subseteq \mathcal{A}$ is a set of attributes. For example, Student$(StudentId, Name, GPA)$ is a schema for a student table.

A **(named) tuple** $t[\bar{A}]$ maps a set of attributes $\bar{A} \subseteq \mathcal{A}$ to the corresponding $D(\bar{A})$. We think of tuples as mapping attribute names to values. This is called the *named perspective*.

An **instance** of a relation, $R(\bar{A})$, is a set of named tuples all having the same schema. We abuse notation and denote the instance and the symbol for a relation in the same way.

Preliminaries A relational symbol R is a valid relational algebra expression.

- The schema of the expression is the schema of R.
- The instance corresponding to the expression R is the set of tuples in the instance of R.

Fundamental Operations We define the five fundamental operations of relational algebra.

- **Selection** is written $\sigma_\theta(q)$ in which q is a relational algebra expression and θ is a Boolean expression over the attributes in the schema of q.
  - The schema of selection is the same as q.
  - The instance corresponding to this expression is defined by the equation:
  $$\sigma_\theta(q) = \{t \in q : \theta(t) \text{ is true }\}$$
- **Projection** is written $\Pi_{\bar{A}}(q)$ in which $\bar{A}$ is a subset of the attributes in the schema of q.

  - The schema of projection is $\bar{A}$.
  - Projection defines an instance as

$$\Pi_{\bar{A}}(q) = \{t[\bar{A}] : t \in q\}$$

    in which $t[\bar{A}]$ corresponds to the attributes in $\bar{A}$.

- **Cross product** is written $q_1 \times q_2$ in which the attributes in q_1 and q_2 are distinct.
  - The schema is the (disjoint) union of the schemata of q_1 and q_2.
  - This expression defines an instance as:

$$q_1 \times q_2 = \{(t_1, t_2) : t_1 \in q_1 \text{ and } t_2 \in q_2\}$$

- **Union** is written $q_1 \cup q_2$ in which q_1 and q_2 have the same schema.
  - The schema is the same as that of q_1 and q_2.
  - This expression defines an instance as:

$$q_1 \cup q_2 = \{t : t \in q_1 \text{ or } t \in q_2\}$$

- **Difference** is written $q_1 - q_2$ in which q_1 and q_2 have the same schema.
  - The schema is the same as that of q_1 and q_2.
  - This expression defines an instance as:

$$q_1 - q_2 = \{t : t \in q_1 \text{ and } t \notin q_2\}$$

Derived and Additional Operations

- *Rename* is written $\rho_{A_1 \to B_1, \ldots, A_n \to B_n}(q)$ in which the schema of q is $A_1, \ldots, A_n$.
  - The schema is $B_1, \ldots, B_n$
  - The instance is unchanged.
  - Rename is a special operator for the named perspective. It is not a derived operator.
- *Natural Join* is written $q_1 \bowtie q_2$.
  - The schema is the union of the schemas of q_1 and q_2.

- The natural join defines a set of tuples:

$$q_1 \bowtie q_2 = \{t : t[\bar{A}] \in q_1 \text{ and } t[\bar{B}] \in q_2\}$$

  In which $\bar{A}$ is the schema of q_1 and $\bar{B}$ is the schema of q_2.

- The natural join is a derived operator; let $\bar{A} \cap \bar{B} = \bar{C}$ be the shared attributes of q_1 and q_2:

$$q_1 \bowtie q_2 = \Pi_{\bar{A} \cup \bar{B}} \left(\sigma_{\bar{C}=D} \left(q_1 \times \rho_{\bar{B} \setminus \bar{C} \to \bar{B} \setminus \bar{C}, \bar{C} \to D}(q_2) \right) \right)$$

- *Intersection* is written $q_1 \cap q_2$, where q_1 and q_2 have the same schema.
  - The schema is the same as the schemas of q_1 and q_2.
  - Intersection defines a set of tuples:

$$q_1 \cap q_2 = \{t : t \in q_1 \text{ and } t \in q_2\}$$

  - Intersection is a derived operator:

$$q_1 \cap q_2 = q_1 - (q_1 - q_2)$$

4 Indexing

[Lectures 13-14]

An index on a file speeds up selections on the search key fields of the index. An index is stored in a separate file (in pages). In particular, there is a data file and there is an index that points into that data file.

Search Key The search key fields:

- Can be any subset of the relation's fields
- Is not the same thing as a key of the relation

Indexes support efficient retrieval of all data entries k* with a particular value k for the search key field.

Extra Material: Page IDs and Record IDs (rids)

A Page ID is a page identifier. A record id (rid for short) are informally called pointers in these notes. *These are **not** exactly like pointers in C.* An rid encodes the location of record unambiguously on secondary storage (e.g., which disk in which file at which point in that file.) In contrast to a C pointer, you can store a pageid or an rid on disk, and it remains valid.

Contents of an Index There are three main options for what the index stores for each k:

1. the actual record
2. (k, rid): the key plus the record id (with duplicates, duplicates are stored).
3. (k, rid list)

Typically only one index uses choice 1, to avoid duplicating the actual records. We will consider only the case of k and a single rid in this course.

4.1 Index Operations

The valid operations on an index are:

- Search: Given a key k find all records that match a key.

- FindRange: Given a pair of keys $[k_1, k_2]$ find records that have keys in this range.
- Insert and Remove (we will not study these in detail)

4.2 Index Classifications

An index is *clustered* if the data is ordered in the same way as the underlying data. Otherwise, we say it is unclustered. Whether the index is clustered or not has an enormous impact on query performance, for *range queries.* The key reason is that when moving from the leaf pages, to the data we will potentially do random IOs and read may pages. This calculation is in the lecture slides!

4.3 B+ Trees

- Very good for range queries and sorted data.
- These are search trees, but the B refers to neither 'Binary' nor 'Balanced' (although B and B+ trees are balanced).
- For B+ trees, the basic idea is to have the leaves of the tree be a linked list of the physical pages. This is to support efficient scanning of a range of values.
- The parameter d of a B+ tree is called its order, which we describe below.
- Both internal and leaf nodes are set to be the size of a single page. See Lectures 13-14 for intuition as to why.

Internal Nodes See the slides for pictures of these structures.

- Each note has $\geq d$ and $\leq 2d$ search keys—except for possibly the root. These search keys are also called *guard entries.*
- Each has a pointer (PageID) to that points to a node for each range of key values that lie between the values of the keys in the node. In particular, if we have two guard entries g_1 and g_2 with a pointer p between then p contains search keys in the range $[g_1, g_2)$.

Leaf Nodes

- Each leaf also has between $\geq d$ and $\leq 2d$ search keys.
- Each search key is paired with a single rid.
- There is also a pageid that points to the next page next leaf node; this way the leaf nodes form a linked list that can be more cheaply traversed for range queries than descending from the root repeatedly.

4.4 The Height of the Tree

Let's give some bounds on how tall the tree is to exercise notation. Suppose we have N leaf pages. We want to compute the height h.

- **Sparsest Tree (Take 1)** Suppose the suppose the root has at least $d+1$ points. Since we know that each internal node has at least $d+1$ pointers. Then, the height of the tree h must satisfy must satisfy $(d+1)^h \geq N$.
- **Sparsest Tree (Take 2)** If the root does not have $d+1$, it must have at least two pointers (else you could simply remove it!). In that case, the minimum number of entries is $2(d+1)^{h-1} \leq N$.
- **Densest Tree**. On the other hand, there are at most $2d+1$ pointers per internal node and $2d+1$ entries per leaf node, hence $N \leq (2d+1)^h$.

  So we have that d, h, and N are in the following relationship.

$$2(d+1)^{h-1} \leq N \leq (2d+1)^h$$

Fill Factor To get a better sense of how tall a tree is in practice, we observe that most nodes are filled between $66-80\%$; we call this percentage the *fill factor*. This room is left to amortize the cost of inserting new data into the tree. This allows us to have intuition about the height of the tree in practice (see the lecture slides for more examples).

- If the leaves are filled with a factor f, this means N pages of records takes N/f pages. So with $f = \frac{2}{3}$ (in general a decent estimate), we need $1.5N$ pages to store these records.
- In fact, nodes at higher levels of the trees tend to be more densely packed than the leaves. Hence, leaf nodes could be filled around 66% (to allow more slack for insertions) while internal nodes are filled to a greater extent.

- This information regarding fill factor specifically is to help with intuition and will not be on the final exam. (The height is fair game).

4.5 Setting d

- d must be small enough that a B+ tree node can fit in one page in memory. On the other hand, we want the node to contain as many nodes as possible to obtain higher fanout.
- All but the last level of tree typically stay in the buffer pool, *which we do not account for in this class.* In some common cases, only I/O is actually needed to fetch a single record. The fanout is $2d+1$ times the fill factor. If the fanout is F, then you have F times more data in the leaves than in the internal nodes. Often, $F \geq 100$, we have orders of magnitude differences in how much data we index on disk versus in memory.
  - A common example of this is the case in which we store all the data (not just the rid) in a leaf node. We won't consider this further.
  - A second example is when all the fields we want are part of the search key; we say the index is covering. E.g., we have an index with search key (A, B, C) and we ask a query like `SELECT B FROM A where A = 10`.
- This is covered explicitly in the lecture slides. However, the order is a function of the size of the key (and the size of a pageid).
- If we store the key and a pageid, the number of entries in a leaf page is a function the size of the key and the size of a page id (to point to the data page).

4.5.1 Search

To search a B+ tree:

- For Search: start at the root and descend down following pointers to the correct leaf. This is illustrated in the slides. As described above, when search for k we go down the first pointer such that $k \in [g_1, g_2)$.
- For a range query $[k_1, k_2]$, we search for k_1 as above. Then we scan along the linked list of leaves for the rest of the range. *This is where we use the pointers described in lecture.*

- The animations explain illustrate how this procedure works see lecture 13. (Animations require powerpoint or video.)

5 External Merge Sort

[Lectures 14-15]

The goal of external sort is to sort a file that is much larger than available memory. This is *the standard merge sort algorithm* adapted to data on disk. In particular, we will count IOs not CPU operations–since this captures the dominant cost in dealing with IO. The algorithms are cheaper according to this metric than say quicksort, which is preferred in an in-memory setting.

Using More Memory In lecture, we showed how to sort an arbitrarily large file with only three buffer pages.[4] We then asked the question: *If we have more than three buffer pages, say $B+1$, how do we use these additional buffer pages to improve the performance of our algorithm?* We used these buffer pages in two ways: (1) longer initial sorted data and (2) merging runs B-way rather than 2 way.

External Merge Sort The input is a single file that contains N pages and our goal is to output that file in sorted order. We also have $B+1$ buffer pages of memory. For concreteness, we will think about sorting the records in ascending order. The algorithm runs in two phases: (1) creating initial runs, and (2) merging those runs. *We will use the merging phase later in sort-merge join.*

- **Phase 1: Create initial runs**. In the first phase of the algorithm, we sort as much data as we can fit into memory. In this case, we can hold $B+1$ buffer pages. We read all N pages but in $B+1$ sized chunks; we sort these chunks in memory (using quicksort, say); and then we write each now-sorted chunk back to the file. We call each such file a *run*, which is a list of records in sorted order. These runs partition the input. The output of Phase 1 is several runs. In more detail,
  - Each run is of size $B+1$, and
  - $\lceil \frac{N}{B+1} \rceil$ runs created in the first step, since the runs form a partition.
- **Phase 2: Merge** The input to this phase is some number of sorted runs, and we merge the sorted runs. We will produce runs that are

[4]Here, the sorting time was $2N(1+\log_2 N)$, in the terminology below our initial runs are of length 1 and we are doing a 2-way merge.

longer/larger, and we will reduce the number of sorted runs on each pass. We will run several passes of merging.[5] The main observation is

> To merge k sorted lists, we only need to hold the first element of each list in memory–no matter how long the list is.

A run is just a list. To find the smallest element in k lists, we just need the smallest element from each list (the first element of the sorted list.) Thus, with $B + 1$ buffer pages, we can perform a B-way merge:

- We use B pages, one for each of the pages, which contains the smallest elements of each run that has not been merged.
- We use one page for output. When the output fills up, we write it to disk.
- The length of the output run produced by merging runs of length $N_1, \dots, N_B$ is $\sum_{i=1}^{B} N_i$, i.e., the sum of the size of the input runs.

See Lecture 12 for more detail on the external merge algorithm which constitutes this merge phase.

Analyzing the number of passes Once we have a run of length N then we've sorted the file! Let's analyze how many passes of the above merging we need to complete this task. We can think about it in two different ways: one by the analyzing the number of runs left after each merge step and one by the length of the runs.

- Method 1: Number of runs.
  - Every merge pass, we reduce the number of runs by a factor of B (we replace them with longer runs!)
  - Thus, to analyze external merge sort, we want to know when we have reduced the number of runs to ≤ 1. Roughly this is when we have done m passes

$$B^{-m} \left\lceil \frac{N}{B+1} \right\rceil \leq 1 \text{ or } \log_B \left\lceil \frac{N}{B+1} \right\rceil \leq m$$

    We pick the smallest value of m that satisfies this inequality, as our goal is to do this in the fewer number of passes.
  - Thus, we need to do $\lceil \log_B \lceil \frac{N}{B+1} \rceil \rceil$ passes to fully merge.

[5]This is described at least twice in lecture with fancy animations, please look at it.

- Method 2: Length of Runs. We will assume for simplicity that all initial runs are the same size, $B + 1$.[6]
  - If we merge B runs of length R, we get a new run of length BR. Thus, after m merge passes we could have runs as large as
$$B^m R$$
  - We have to repeat this process until we have one run that is the length of the entire file. Thus, ignoring ceilings, after m merge passes we need to solve:
$$B^m(B+1) \geq N$$
    Thus, $m \geq \log_B \lceil \frac{N}{B+1} \rceil$ and again we need to do $\lceil \log_B \lceil \frac{N}{B+1} \rceil \rceil$ passes.

IO Cost The key observation is that in both phases on each pass, *each page is read once and written exactly once for 2 IOs per page.* Thus, our number of IO operations is $2N(1 + m)$ where 1 is the initial pass and m is the number of merge passes. In other words our total IO cost is:

$$2N\left(1 + \lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \rceil\right)$$

Further optimization: Repacking In Lecture 13 we outline a further optimization where we "merge" in the buffer as we sort at the initial stage, to create longer initial runs. As a rough estimate, using this technique we can get initial runs of size $\approx 2(B + 1)$ and then our total IO cost is:

$$2N\left(1 + \lceil \log_B \left\lceil \frac{N}{2(B+1)} \right\rceil \rceil\right)$$

Note that since this improvement is approximate, we'll explicitly say if repacking is being used (unless we do, in any of the calculations below, assume it is not)

[6]If you do not make this assumption, the length of the merged file is the size of the sum of the input files. The details are not too nasty. In particular, for m, all but one run is of length $B^m(B + 1)$. That is, there is exactly one run that is smaller in any pass.

6 Join Algorithms

We described three join algorithms: block nested loop join (see slides), Hash Join, and Sort Merge Join. We will only talk about equijoins here, so it makes sense to talk about a join key. For example, we could use it on a query like the following:

```
SELECT * FROM Employee e, Dept d where e.did = d.id
```

Or more abstractly.

```
SELECT * FROM R, S where R.A = S.A
```

We will use A as the join key throughout (although it could be a conjunction of more than one equality atoms e.g., `R.A = S.A AND R.B = S.B`).

6.0.1 Notation

We'll use the following notation as in lecture; for a relation R:

- $P(R)$ is the number of *pages* or *blocks* that make up R [7]
- $T(R)$ is the number of tuples in R
- OUT represents the IO cost to write the output of the join back to disk. In the worst case OUT $= \frac{T(R)T(S)}{P}$ where P is the number of tuples that fit on an output page, however in general OUT is much closer to $O(P(R) + P(S))$. Either way, **OUT will be the same for any join algorithm**, so is not important to any comparisons we do

6.1 Block Nested Loop Join (BNLJ)

You should know the cost equations for nested loop join (NLJ) and block nested loop join (BNLJ). It is implicitly used below. If you have $B+1$ buffer pages, and you join relation R (having $P(R)$ pages) and relation S (having $P(S)$ pages). This algorithm is described in Algorithm **??**

Memory We allocate our buffers as follows:

- We read in chunks R in size $B - 1$.
- We read in one page of S one at a time using 1 buffer page.
- We have 1 page for output. As it fills up, we write it out to disk.

[7] Again, in this class we will conflate *page* and *block*, and use the former in this section

```
BNLJ(R : Relation,S : Relation, B : Integer);
**Data:** Relation $R$, a Relation $S$, and $B+1$ buffer pages.
**Result:** Compute $R \bowtie S$
Setup a single output page;
foreach Block of $B-1$ pages of $R$ do % $B$ blocks read at a time
    foreach page $P_S$ of $S$ do % this is a read from disk
        foreach tuple $s \in P_S$ do % The part below is in memory
            foreach tuple $r$ in those $B-1$ pages do
                if $r[A] = s[A]$ then Write $(r, s)$ to the output page;
                if output page is full then Write output page to disk;
            end
        end
    end
end
if Output page is not empty then Write the output page to disk;
```

Algorithm 1: Block Nested Loop Join

IO Cost The cost of this algorithm is:

$$P(R) + \left\lceil \frac{P(R)}{B-1} \right\rceil P(S) + \text{OUT}$$

Notice that this formula is not symmetric. You should be aware of how to minimize it.

- Notice that if $P(R) < B-1$ then BNLJ has IO cost $P(R) + P(S) +$ OUT, i.e. it's linear

- BNLJ can be used with essentially arbitrary join predicates, e.g., inequality join conditions such as $R.A > S.B+5$ or $f(R.A, S.B)$ where f is an arbitrary boolean function. So it's very useful. In contrast, hash and sort-merge join are for equijoins (joining on equality constraint(s) only) and natural joins (joining on equality of all shared attributes). On the one hand this limits what joins what we can do, however we can be much smarter about these ones!

- Equijoins are more common. For the rest of this section, we can think of the query $R(A, B) \bowtie S(A, C)$.

6.2 Index Nested Loop Join

We use an index to avoid scanning the entire relation as shown in Algorithm **??**. We assume that we have an index on S with search key that contains (at least) A. The algorithm is as follows:

```
foreach tuple of R do % Read R one page at a time
    Search using r[A] into the index on S;
    Output all matches returned by the index;
end
```

Algorithm 2: Index Nested Loop Join

The cost of this algorithm is as follows: we read in each page of R exactly once, so this incurs a total cost of $P(R)$. For each such tuple, we perform one index lookup in S. Thus, our cost can be written:

$$P(R) + T(R)\text{IndexLookup(S)} + \text{OUT}$$

Some comments on the algorithm:

- In our simplified model, the IndexLookup(S) is proportional to the height of the tree, and so the number of leaf pages in S. *This level of detail is all you are required to know.*

- This algorithm requires three buffer pages to run: One for the input, one the output, and one for the index page. However, in practice, we can use the extra buffer pages to facilitate obvious caching opportunities. For example, e.g., if the same value of $r[A]$ is queried twice, we could have cached all pages to the leaf in the index, which could fit in the buffer.

6.3 Sort Merge Join

The simple algorithm is roughly as follows: *sort using the external merge sort, intersect the resulting lists, and output all matches.*

The sort phase is just external merge sort run on each of the relations.

Now, we have two relations R and S sorted by the join key A. For a tuple $r \in R$, let $r[A]$ denote the value of the join key. Similarly, let $s[A]$ denote the value of the join key for a tuple $s \in S$.

Merge For Join We run essentially the same merge algorithm but now over two files. Our goal is to find matching keys (and discard non matching tuples). In essence, we are computing a list intersection with sorted lists. All comparisons are with respect to the A value. That is,

- Find the minimum element of R in A-order, call it r.
- Find the minimum element of S in A-order, call it s.
- If $r[A] < s[A]$, then we know that no tuple remaining in S can possibly match with r. In this case, we advance to the next tuple in in r. (If $r[A] > s[A]$, then the symmetric statements hold.)
- If $r[A] = s[A]$, then we have found a match. We write it to the output buffer.
  - If there are multiple tuples with the same join key in R, S or both, then we need to output the cross product of all of these tuples. We find all tuples in R and S with join key equal $r[A]$ (and so also s).
  - Notice this cross product could be a huge number of tuples. In the worst case, *we could even get an entire cross product as output.*
- We then advance through each list.
- **Backup:** Note that if there are many duplicate join keys, we may have to "back up" and read in tuples we've already read in again. See the animation in Lecture 15.

This is just the merge algorithm with a small modification. We are not describing any of the bookkeeping for merging a list, but it should be straightforward how it works (and you have animations from lecture). To do this merge, we only need three pages. We use one page of R and one page of S and use the algorithm from lecture.

The Cost With this simple algorithm to join R and S our cost is:

$$\underbrace{2P(R)\left(1 + \lceil \log_B \left\lceil \frac{P(R)}{B+1} \right\rceil \rceil \right) + 2P(S)\left(1 + \lceil \log_B \left\lceil \frac{P(S)}{B+1} \right\rceil \rceil \right)}_{\text{sorting cost}} + \underbrace{P(R) + P(S)}_{\text{merging cost}} + \text{OUT}$$

Note here we're explicitly writing OUT. How large can it be? Note also that due to backup (mentioned above), the merging cost could actually be larger (but usually won't be).

6.3.1 Optimization: Join and Merge!

We can do a simple optimization to take advantage of having more memory in the final merge phase. In some cases, this may shave off an entire pass over the data. If the buffer is large enough, then we can skip the last merge step for the sorts of R and S individually. Instead, we can do their final merge step and the join at the same time. In particular, with $B + 1$ buffer pages, if the sum of the number of runs from both R and S is less than than B,[8] we simply run the merge step for each in parallel. In particular, we bring in one page from each of the runs in both R and S, and perform the merge to find the smallest tuple (and run the merging step of the join algorithm above). This shaves off the pass we would spend writing out the fully sorted versions of R and S, only to merge the two lists again.

So under what conditions does this stroke of good luck occur to let us sort in two passes? Suppose that we have $B + 1$ buffer pages. We just plug in the number of runs we would have using the standard sorting algorithm. The number of runs satisfy the above condition when:

$$\left\lceil \frac{P(R)}{B+1} \right\rceil + \left\lceil \frac{P(S)}{B+1} \right\rceil \leq B$$

In particular, very roughly, as long as $P(R) + P(S) \leq B^2$ we can do the entire sort merge join in two passes. We can do slightly better below to recover the statement that $\max\{P(R), P(S)\} \leq B^2$.

Further Optimization You can see that we can play similar games to save one pass over R or one pass over S on the final merge run. It makes the cost formula nastier, although it should be easy to see that you could write code to estimate this formula.

[8]Note we need one page for output as is standard.

Repack: Optimization for longer runs It turns out one can create initial sorted runs of length $\approx 2(B+1)$ using what we will call the *repack optimization*. From the above inequality, this means that you finish in two passes when

$$\frac{P(R)+P(S)}{2} \leq \max\{P(R), P(S)\} \leq B^2$$

This factor of 2 is helpful to make the comparison to hashing easier.

6.4 Hash Join

The first phase of the Hash Join (HJ) algorithm is to partition into small enough buckets that the buckets from a smaller relation can fit in memory. A bucket is defined operationally: *all the values in the bucket will have the same hash value for the join key A.* In particular, the buckets form a partition of the input file so that if two tuples have the same join key, they will go to the bucket.

6.4.1 Partitioning Phase

We first describe the partitioning phase in Algorithm **??**.

Setup We assume we have an infinite family of hash functions $h_1, h_2, \dots,$ that are distinct.[9] A **bucket** is a file that consists of a set of pages. Only one page from a bucket will be in memory at any one time in Algorithm **??**.

Partitioning Pass Given an input relation R with $P(R)$ pages and a size bound T, produce buckets so that each bucket contains no more than T pages.[10] The main operation in the partitioning phase is given $B+1$ buffer pages partition the input data into B buckets.

- We hash each tuple $r \in R$ on $r[A]$. This implies that all tuples with the same value of A are in the same bucket.
- All values in a bucket have the same of value the hash; this is the definition of a hash bucket.

[9] We can be precise about, but it's painful. The idea is that two hash functions will hash different tuples to different places.

[10] This may not always be possible, if for example all values are the same. Let's ignore this very minor detail that may obscure understanding. We will assume that the number of distinct values fit on T pages.

```
Partition($R, A, \bar{h}, T$);
Data: Relation $R$, an attribute of $R$ called $A$, a family of hash
      functions $\bar{h}$, and a bucket size $T$.
Result: A set of buckets. Each bucket contains no more than $T$
        pages
Buckets $\leftarrow \{R\}$. % initially the file is a single bucket;
$i \leftarrow 1$ % this is which hash function to use;
while There is a Bucket with more than $T$ pages do
    foreach Bucket $P$ in Buckets do
        Create $B$ new buckets $P_1$ to $P_B$ % each one will have one page
          in buffer.;
        foreach tuple $t \in P$ do % Read 1 page of bucket $P$ at a time
            Let $j = h_i(t[A])$ % Use the hash function for this round.;
            Add the tuple $t$ to the buffer page for bucket $P_j$;
            if If the buffer page for $P_j$ is full then
                Write $P_j$ to disk;
                Clear contents of memory $P_j$ in memory
            end
        end
        Write out all non-empty buckets.;
        Add new partitions to the set of all partitions.
    end
    $i \leftarrow i + 1$ % Advance to the next round.
end
```

Algorithm 3: Hash Partition

- We use 1 buffer page for input and B buffer pages as output (one page per *bucket*). As the output buckets fill up, we write them to disk. As a result, a bucket may be of size up to $P(R)$ (if the data are highly skewed).

- If we assume "no skew", operationally this means that each bucket will be of roughly equal size after hashing. We will assume there is no skew. Hence, the number of pages in each bucket under this idealized assumption is $\lceil \frac{P(R)}{B} \rceil$. We make this *uniform hashing or no skew assumption below.*

We can repeat the partitioning above to further partition the data. In the next phase, we take each input bucket one at a time and treat it as a new input. That is, we partition the data using a new hash function h_2. Each bucket produces B new buckets, hence The output is a set of B^2 buckets. We can repeat this process several times. After m passes, this process creates B^m buckets.

- Observe that on each pass, we use a distinct hash function.[11] We use hash function h_i on round i. Note that two tuples s, t are in the same bucket after m passes if:

  $$h_i(s[A]) = h_i(t[A]) \text{ for } \textbf{all } i = 1, \ldots, m$$

  Alternatively, for two tuples s and t if there is even one i such that $h_i(t[A]) \neq h_i(s[A])$ for $i \in \{1, \ldots, m\}$ then, s and t hash to different buckets.

- We also hope the size of the bucket goes down: *here is where we use the distinct hash function assumption.* If we reused the same hash function, the size of a bucket would not decrease after the first round. If we hash with a different hash function h_2, however, then this allows us to further split the bucket. As a result, the size of each bucket will again go down by a factor of B. We repeat this process, and we use a distinct hash function on each pass.

How many passes? Notice after each pass, we introduce a factor of B more buckets and the size of each bucket is reduced by a factor of B (look familiar?). Hence, if we start with $P(R)$ pages of data, after m passes the size of a bucket is: $B^{-m}P(R)$.

- To ensure that $B^{-m}P(R) \leq T$, we again need to find the smallest integer such that this holds which is :

  $$m = \left\lceil \log_B \frac{P(R)}{T} \right\rceil \tag{1}$$

- Observe that in each pass, we read and write each page in the relation exactly once. Thus, in m passes, the partitioning phase requires $2P(R)m$ IOs.

- The assumption of no skew is likely not met in practice, which is why we use statistics to estimate the skew (e.g., histograms). This is a critical point and where our simplified analysis breaks down: without

[11]If we use the same hash function, the bucket size would not reduce.

this hash join looks better than sort merge join in *nearly* all cases in our simple analysis. (A caveat is if we need the data in sorted order for a downstream operation, e.g., an `ORDER BY`, or if the data are already sorted, e.g., the data are stored in a clustered index.)

- We create a separate function `PartitionTimes`$(R, A, \bar{h}, m)$ in which we replace the termination condition of the while loop of Algorithm **??** with simply running m times. In this case, we produce B^m buckets.

```
Data: Two relations $R$ and $S$ both having attribute $A$
Result: The Join of $R \bowtie S$ on $A$
Buckets_of_R ← Partition($R, A, \bar{h}, B-1$);
% Each bucket of $R$ occupies $B-1$ pages;
Suppose $|Buckets\_of\_R| = B^m$;
Buckets_of_S ← PartitionTimes($S, A, \bar{h}, m$);
foreach Bucket $B_R$ in Buckets_of_R do
    Let $B_S$ be the corresponding joinable bucket to $B_R$;
    BNLJ($B_R$, $B_S$, $B$). % Runs in 1 pass, since $B_R$ fits in memory!
end
```

Algorithm 4: Hash Join

6.4.2 Join Phase

The hash join algorithm works as follows. Given two relations R and S, we partition both of them until the buckets for the smaller relation fit in memory. As we will see, *only the buckets for the smaller relation need to fit in memory to run the join.* Then we perform (effectively) a block nested loop join in which the entire smaller relation fits in memory.

- First, partition each R and S on the join key A until the buckets produced by *either* R or S is "small". Below, we will see that "small" means $B-1$ given $B+1$ buffer pages. Let m be the number of passes.

- For each of the m passes, we use the same function for R and S. That is in phase i, we use h_i for *both* relations. This ensures that if two tuples agree on the join key, then they will be in the *same bucket* after partitioning.

Now the join phase. We have partitioned R into buckets and S into buckets, and now we want to compute the output.

- The key property of the partitioning is that since we use the same family of hash functions on each pass, we can select pairs of buckets B_R from R and B_S from S such that for any tuples $r \in B_R$ and $s \in B_S$, we have
$$h_i(r[A]) = h_i(s[A]) \text{ for each } i = 1, \ldots, m.$$
  Notice that this holds for exactly one pair of buckets. We call such pairs of buckets **joinable**. In particular, *if r and s can join, then they are in joinable buckets.* Not all tuples will have the same value of A in a bucket, so not all tuples in the buckets participate in a join.

- For each such pair of joinable buckets, we can run BNLJ. More directly, we compute B_R joined with B_S. Note that we use our $B+1$ buffer pages as follows:
  - $B-1$ pages for B_R
  - 1 page to hold a single page B_S at a time.
  - 1 page to hold the output.

  Since we have $|B_R| \leq B-1$ from the partitioning phase, we hold the whole block B_R in memory and scan through B_S one page at a time. We then compute whether each pair of tuples in these blocks match. In IOs, this costs the number of pages in B_R plus the number of pages in B_S. Since these are partitions and joinable buckets in a $1-1$ relationship, the total cost of this phase is $P(R)+P(S)+\text{OUT}$.

Cost Assume $P(R) \leq P(S)$ w.l.o.g., and that we have $B+1$ buffer pages. The goal is to partition the data until the buckets of R (the smaller relation) are less than size $B-1$ (i.e., take $T = B-1$ in Equation **??**). By our partitioning phase above, we require m passes where:

$$m = \lceil \log_B \left\lceil \frac{P(R)}{B-1} \right\rceil \rceil$$

Recall that in each pass, we read and write each page in each relation exactly once. Thus, the entire partitioning phase costs $2m(P(R)+P(S))$ IOs. Finally, we need to do one final pass over the data for the join phase, which takes a single pass. Thus, our total cost in IOs is:

$$\underbrace{2\lceil \log_B \left\lceil \frac{P(R)}{B-1} \right\rceil \rceil (P(R)+P(S)) +}_{\text{Partitioning Steps}} \underbrace{P(R)+P(S)}_{\text{Final Merge Step}} + \text{OUT}$$

6.5 Join Algorithm Comparison

- The memory requirement to complete in m passes is proportional to the number of pages in the larger relation in sorting (since both relations must be sorted). In contrast, in hashing the memory requirement is proportional to the size of the minimum relation.
- Skew is a major challenge for hashing. In the worst case, everything hashes to a single bucket! This is the point of the histogram and `ANALYZE` questions.

7 Histograms

We discuss equiwidth and equidepth histograms in Lecture 17, and how to use these structures to make estimates of the number of tuples in queries. **Make sure you understand these structures!**

- The cost formulas above rely on things like skew- we need to estimate that. If we have more than one join, we need to estimate its output size.
- Maintaining accurate statistics seems easy but is fairly tricky: we want to do so in the smallest amount of space we can!
- Estimation with histograms, this is done in Lecture 17 and you should know how to do it.

Example 7.1 (Estimating Join Size with a Histogram) *Consider the join $R(A, B) \bowtie S(A, C)$. Consider a pair of buckets for the values of A in R and the values of A in S.*

- *Suppose one bucket for $R[A]$ spans* 5 *values with frequency* 10*, we estimate that each value occurs twice* $(10/5 = 2)$.
- *Suppose one bucket for $S[A]$ spans* 6 *values with frequency* 18*, we estimate that each value occurs three times* $(18/6 = 3)$.
- *Suppose these buckets overlap on* 4 *values on A, then we estimate* $(2 * 3) * 4 = 24$ *values produced by this pair of buckets in the join.*

To obtain an estimate for the entire output, we sum the contributions of each pair of buckets to produce the estimate for the join. However, some pairs of buckets will not overlap at all, and so will contribute 0 *to the output.*

An alternate strategy, is that we could also write it like this:

$$|R(A,B) \bowtie S(A,C)| = \sum_{a \in \mathrm{D}(A)} |\sigma_{A=a} R||\sigma_{A=a} S|$$

here $\mathrm{Dom}(A)$ *is the domain of values for* A*. Using our standard histogram algorithm from class, we can estimate* $|\sigma_{A=a} R|$ *for any value* a*.*

The first algorithm above just uses the buckets, which may be more efficient if the buckets are large (which is not a major concern). The reason to introduce it is to make sure you know how to operate on the buckets themselves.