

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 17th November, 2015 at 14:38

Contents

1	Transactions	4
1.1	The ACID Properties	4
1.2	TXNs In SQL	5
1.3	Logging for Atomicity & Durability	5
1.3.1	The Log	5
1.3.2	Write-Ahead Logging	5
1.4	Concurrency with Isolation & Consistency	6
1.4.1	Scheduling Definitions	6
1.4.2	Anomalies	6
1.4.3	Conflicts	6
1.4.4	Locking	6
1.4.5	Two-Phase Locking (2PL) Protocol	7
1.4.6	Conflict Equivalence	7
1.4.7	Dependency Graphs	7
1.4.8	Deadlock	8
1.5	Some Highlights and Tips	9
2	Storage	11
2.1	Disk Mechanics	11
2.2	Disk Latency	11
2.3	High-level: Disk vs. Main Memory	12
2.4	Cost Model	12
3	Relational Algebra	13
4	Indexing	16
4.1	Index Operations	16
4.2	Index Classifications	17
4.3	B+ Trees	17
4.4	The Height of the Tree	18
4.5	Setting d	19
4.5.1	Search	19
5	External Merge Sort	21
6	Join Algorithms	24
6.0.2	Notation	24
6.1	Block Nested Loop Join (BNLJ)	24
6.2	Index Nested Loop Join	26

6.3	Sort Merge Join	27
6.3.1	Optimization: Join and Merge!	28
6.4	Hash Join	29
6.4.1	Partitioning Phase	29
6.4.2	Join Phase	33
6.5	Join Algorithm Comparison	35
7	Histograms	35

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

- **Atomicity:** The DBMS state either reflects all effects of a transaction, or none of them.
- **Consistency:** If (explicit and implicit) integrity constraints hold over the state before a transaction, they will also hold over the state after the transaction
- **Isolation:** The effect of transactions concurrently is the same as the effect of running all transactions one-at-a-time
- **Durability:** 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

28 1.2 TXNs In SQL

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

32 1.3 Logging for Atomicity & Durability

33 1.3.1 The Log

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

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

44 1.3.2 Write-Ahead Logging

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

54 1.4 Concurrency with Isolation & Consistency

55 1.4.1 Scheduling Definitions

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

65 1.4.2 Anomalies

- 66 • Lost update/Overwriting uncommitted data (Write after Write)
- 67 • Dirty read (Read after Write; writing transaction subsequently aborts)
- 68 • Inconsistent read (A task sees some but not all changes made by an-
69 other)

70 1.4.3 Conflicts

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

77 1.4.4 Locking

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

79 1.4.5 Two-Phase Locking (2PL) Protocol

- 80 • In 2PL, there are two phases of locking: lock acquisition and lock
81 release. Once TXN releases a lock, it cannot acquire any more locks.
- 82 • Each transaction must obtain a *S (shared) lock* on an object before
83 reading it, and a *X (exclusive) lock* on an object before writing it
 - 84 – An S lock can be held by multiple TXNs
 - 85 – An X lock can be held by only one TXN
 - 86 – An X lock cannot be held on an object at the same time as any
87 other S or X locks on that object
- 88 • All locks held by a transaction are released when the transaction com-
89 pletes.
- 90 • We use Strict 2PL in this course, which has the added requirement over
91 2PL that a transaction releases locks only at the end of the transaction.

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

95 1.4.6 Conflict Equivalence

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

98 Two schedules are *conflict equivalent* if:

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

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

104 1.4.7 Dependency Graphs

105 A dependency graph consists of a directed graph with:

- 106 • 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 agrees with the edges to create a serial schedule. Notice that S is conflict equivalent to the serial schedule, since each conflicting action is 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 exclusive 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.

¹This 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 deadlock²
- 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.

131

132 1.5 Some Highlights and Tips

133 Note there are examples in the Lectures 8-9. Check them out! Serializability
134 is the database gold standard notion of correctness. It allows us to inter-

135 leave the order of transactions for performance. Conflict serializability is a
136 restricted notion that captures what our locking protocol allows.

- 137 • Intuitively, to see if a schedule is equivalent to a fixed serial schedule
138 transactions- say T_1, \dots, T_k - you simulate running these transactions.
139 You check that for each object they write, the schedule and the serial
140 schedule have the *same value at the end of both schedules*. It does
141 not matter if the two databases have different values in the middle of
142 execution!
- 143 • To test if a schedule is *serializable*, you can do the following. Sup-
144 pose the schedule contains k transactions, T_1, \dots, T_k . You consider
145 whether that schedule is equivalent to *some* ordering of the transac-
146 tions. You simply try the algorithm from the preceding bullet point on
147 each possible orders (all $k!$ orders). This is definitely a tougher notion
148 to test!
- 149 • If no such ordering exists, the transaction is *not* serializable. Intu-
150 itively, it means that there is some kind of dependence like in Lecture
151 12, where some examples illustrate this.
- 152 • To test if a schedule is *conflict serializable*, you simply run the strict
153 2PL locking protocol and see if you get stuck! More precisely, you
154 form the conflict graph and check if it is acyclic.
- 155 • Not all serializable schedules are conflict serializable. This makes sense
156 as conflict Serializability is a more strict notion than serializability!
- 157 • You could still have a deadlock even when running Strict 2PL. If you
158 don't have a deadlock, and you run strict 2PL to completion for a
159 schedule, then your schedule is serializable. This captures the sense in
160 which locking is correct.

161 2 Storage

162 [Lecture 12]

163 We study hard disks, which we assume are durable (we do not worry
164 about failure, which is handled by other mechanisms). Disks are block
165 devices.³ This means we can only read or write data at block sizes (4K,
166 8K or so). We need to bring the data into memory to modify it or have
167 the CPU read it, a region of memory called the buffer. The buffer may be
168 smaller than all available memory.

169 2.1 Disk Mechanics

170 **These are not important for the exam.**

- 171 • Disk storage hardware is comprised of multiple *platters*.
- 172 • A *track* is a ring in a platter (bytes per track on the order of 10^5)
- 173 • A *cylinder* is one track from every platter (the track in the same
174 location on each platter)
- 175 • Read and write from the disk in units of disk blocks (typically 4K, 8K,
176 16K)

177 2.2 Disk Latency

178 **These are not important for the exam.**

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

³We will conflate blocks and pages for this course.

185 2.3 High-level: Disk vs. Main Memory

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

196 2.4 Cost Model

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

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

209 3 Relational Algebra

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

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

216 A **schema** for a relation is a symbol name and a set of attributes written
 217 $R(\bar{A})$ in which R is the symbol name for the relation, and $\bar{A} \subseteq \mathcal{A}$ is a set of
 218 attributes. For example, $\text{Student}(\text{StudentId}, \text{Name}, \text{GPA})$ is a schema for
 219 a student table.

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

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

226 **Preliminaries** A relational symbol R is a valid relational algebra expres-
 227 sion.

- 228 • The schema of the expression is the schema of R .
- 229 • The instance corresponding to the expression R is the set of tuples in
 230 the instance of R .

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

- 233 • **Selection** is written $\sigma_\theta(q)$ in which q is a relational algebra expression
 234 and θ is a Boolean expression over the attributes in the schema of q .
 - 235 – The schema of selection is the same as q .
 - 236 – The instance corresponding to this expression is defined by the
 237 equation:

$$\sigma_\theta(q) = \{t \in q : \theta(t) \text{ is true} \}$$

- 238 • **Projection** is written $\Pi_{\bar{A}}(q)$ in which \bar{A} is a subset of the attributes
 239 in the schema of q .

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

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

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

- 243 • **Cross product** is written $q_1 \times q_2$ in which the attributes in q_1 and q_2
- 244 are distinct.

- 245 – The schema is the (disjoint) union of the schemata of q_1 and q_2 .
- 246 – This expression defines an instance as:

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

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

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

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

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

253 **Derived and Additional Operations**

- 254 • *Rename* is written $\rho_{A_1 \rightarrow B_1, \dots, A_n \rightarrow B_n}(q)$ in which the schema of q is
- 255 A_1, \dots, A_n .
- 256 – The schema is B_1, \dots, B_n
- 257 – The instance is unchanged.
- 258 – Rename is a special operator for the named perspective. It is not
- 259 a derived operator.
- 260 • *Natural Join* is written $q_1 \bowtie q_2$.
- 261 – The schema is the union of the schemas of q_1 and q_2 .

262 – The natural join defines a set of tuples:

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

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

264 – The natural join is a derived operator:

$$q_1 \bowtie q_2 = \Pi_{\bar{A} \cup \bar{B}} \left(\sigma_{\{(t_1, t_2) : t_1[\bar{A} \cap \bar{B}] = t_2[\bar{A} \cap \bar{B}]\}} (q_1 \times q_2) \right)$$

265 Where here we relax the definition of the cross product to allow
 266 for non-disjoint schema (the actual derivation just requires using
 267 renames to get around this)

268 • *Intersection* is written $q_1 \cap q_2$, where q_1 and q_2 have the same schema.

269 – The schema is the same as the schemas of q_1 and q_2 .

270 – Intersection defines a set of tuples:

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

271 – Intersection is a derived operator:

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

272 4 Indexing

273 [Lectures 13-14]

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

277 **Search Key** The search key fields:

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

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

282

Extra Material: Page IDs and Record IDs (rids)

283 A Page ID is a page identifier. A record id (rid for short) are in-
formally 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.

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

- 286 1. the actual record
- 287 2. (k, rid) : the key plus the record id (with duplicates, duplicates are
288 stored).
- 289 3. $(k, \text{rid list})$

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

292 4.1 Index Operations

293 The valid operations on an index are:

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

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

298 4.2 Index Classifications

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

305 4.3 B+ Trees

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

316 **Internal Nodes** See the slides for pictures of these structures.

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

323 Leaf Nodes

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

329 4.4 The Height of the Tree

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

- 332 • **Sparsest Tree (Take 1)** Suppose the suppose the root has at least
333 $d + 1$ pointers. Since we know that each internal node has at least $d + 1$
334 pointers, and Then, the height of the tree h must satisfy must satisfy
335 $(d + 1)^h \leq N$.
- 336 • **Sparsest Tree (Take 2)** If the root does not have $d + 1$, it must have
337 at least two pointers (else you could simply remove it!). In that case,
338 the minimum number of entries is $2(d + 1)^{h-1} \leq N$.
- 339 • **Densest Tree.** On the other hand, there are at most $2d + 1$ pointers
340 per internal node and $2d + 1$ entries per leaf node, hence $N \leq (2d + 1)^h$.
- 341 So we have that d , h , and N are in the following relationship.

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

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

- 347 • If the leaves are filled with a factor f , this means N pages of records
348 takes N/f pages. So with $f = \frac{2}{3}$ (in general a decent estimate), we
349 need $1.5N$ pages to store these records.
- 350 • In fact, nodes at higher levels of the trees tend to be more densely
351 packed than the leaves. Hence, leaf nodes could be filled around 66%
352 (to allow more slack for insertions) while internal nodes are filled to a
353 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.)

389 5 External Merge Sort

390 [Lectures 14-15]

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

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

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

408 • **Phase 1: Create initial runs.** In the first phase of the algorithm,
409 we sort as much data as we can fit into memory. In this case, we can
410 hold $B+1$ buffer pages. We read all N pages but in $B+1$ sized chunks;
411 we sort these chunks in memory (using quicksort, say); and then we
412 write each now-sorted chunk back to the file. We call each such file
413 a *run*, which is a list of records in sorted order. These runs partition
414 the input. The output of Phase 1 is several runs. In more detail,

415 – Each run is of size $B+1$, and
416 – $\lceil \frac{N}{B+1} \rceil$ runs created in the first step, since the runs form a parti-
417 tion.

418 • **Phase 2: Merge** The input to this phase is some number of sorted
419 runs, and we merge the sorted runs. We will produce runs that are

⁴Here, 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 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.

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

449 • Method 2: Length of Runs. We will assume for simplicity that all
 450 initial runs are the same size, $B + 1$.⁶

451 – If we merge B runs of length R , we get a new run of length BR .
 452 Thus, after m merge passes we could have runs as large as

$$B^m R$$

453 – We have to repeat this process until we have one run that is the
 454 length of the entire file. Thus, ignoring ceilings, after m merge
 455 passes we need to solve:

$$B^m(B + 1) \geq N$$

456 Thus, $m \geq \log_B \lceil \frac{N}{B+1} \rceil$ and again we need to do $\lceil \log_B \lceil \frac{N}{B+1} \rceil \rceil$
 457 passes.

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

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

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

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

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

⁶If 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.

469 6 Join Algorithms

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

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

475 Or more abstractly.

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

477 We will use A as the join key throughout (although it could be a con-
478 junction of more than one equality atoms e.g., $R.A = S.A$ AND $R.B = S.B$).

479 6.0.2 Notation

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

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

488 6.1 Block Nested Loop Join (BNLJ)

489 You should know the cost equations for nested loop join (NLJ) and block
490 nested loop join (BNLJ). It is implicitly used below. If you have $B+1$ buffer
491 pages, and you join relation R (having $P(R)$ pages) and relation S (having
492 $P(S)$ pages). This algorithm is described in Algorithm 1

493 **Memory** We allocate our buffers as follows:

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

⁷Again, 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 PS of S do % this is a read from disk
        foreach tuple s ∈ PS 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
end
if Output page is not empty then Write the output page to disk
Algorithm 1: Block Nested Loop Join

```

497 **IO Cost** The cost of this algorithm is:

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

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

- 500 • Notice that if $P(R) < B - 1$ then BNLJ has IO cost $P(R) + P(S) +$
501 OUT, i.e. it's linear
- 502 • BNLJ can be used with essentially arbitrary join predicates, e.g., in-
503 equality join conditions such as $R.A > S.B + 5$ or $f(R.A, S.B)$ where f
504 is an arbitrary boolean function. So it's very useful. In contrast, hash
505 and sort-merge join are for equijoins (joining on equality constraint(s)
506 only) and natural joins (joining on equality of all shared attributes).
507 On the one hand this limits what joins what we can do, however we
508 can be much smarter about these ones!
- 509 • Equijoins are more common. For the rest of this section, we can think
510 of the query $R(A, B) \bowtie S(A, C)$.

511 6.2 Index Nested Loop Join

512 We use an index to avoid scanning the entire relation as shown in Algo-
 513 rithm 2. We assume that we have an index on S with search key that
 514 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

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

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

518 Some comments on the algorithm:

- 519 • In our simplified model, the $\text{IndexLookup}(S)$ is proportional to the
 520 height of the tree, and so the number of leaf pages in S . *This level of*
 521 *detail is all you are required to know.*
- 522 • This algorithm requires three buffer pages to run: One for the input,
 523 one the output, and one for the index page. However, in practice, we
 524 can use the extra buffer pages to facilitate obvious caching opportuni-
 525 ties. For example, e.g., if the same value of $r[A]$ is queried twice, we
 526 could have cached all pages to the leaf in the index, which could fit in
 527 the buffer.

528 6.3 Sort Merge Join

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

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

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

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

- 539 • Find the minimum element of R in A -order, call it r .
- 540 • Find the minimum element of S in A -order, call it s .
- 541 • If $r[A] < s[A]$, then we know that no tuple remaining in S can possibly
542 match with r . In this case, we advance to the next tuple in r . (If
543 $r[A] > s[A]$, then the symmetric statements hold.)
- 544 • If $r[A] = s[A]$, then we have found a match. We write it to the output
545 buffer.
 - 546 – If there are multiple tuples with the same join key in R , S or
547 both, then we need to output the cross product of all of these
548 tuples. We find all tuples in R and S with join key equal $r[A]$
549 (and so also s).
 - 550 – Notice this cross product could be a huge number of tuples. In the
551 worst case, *we could even get an entire cross product as output.*
- 552 • We then advance through each list.
- 553 • **Backup:** Note that if there are many duplicate join keys, we may
554 have to "back up" and read in tuples we've already read in again. See
555 the animation in Lecture 15.

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

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

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

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

565 6.3.1 Optimization: Join and Merge!

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

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

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

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

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

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

588 **Repack: Tournament Sort for Longer Runs** It turns out one can
 589 create initial sorted runs of length $\approx 2(B+1)$ using *tournament sort*.⁹ From
 590 the above inequality, this means that you finish in two passes when

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

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

592 6.4 Hash Join

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

599 6.4.1 Partitioning Phase

600 We first describe the partitioning phase in Algorithm 3.

601 **Setup** We assume we have an infinite family of hash functions h_1, h_2, \dots ,
 602 that are distinct.¹⁰ A **bucket** is a file that consists of a set of pages. Only
 603 one page from a bucket will be in memory at any one time in Algorithm 3.

604 **Partitioning Pass** Given an input relation R with $P(R)$ pages and a size
 605 bound T , produce buckets so that each bucket contains no more than T
 606 pages.¹¹ The main operation in the partitioning phase is given $B+1$ buffer
 607 pages partition the input data into B buckets.

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

⁹http://en.wikipedia.org/wiki/Tournament_sort

¹⁰We can be precise about, but it's painful. The idea is that two hash functions will hash different tuples to different places.

¹¹This 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

- 612 • We use 1 buffer page for input and B buffer pages as output (one page
613 per *bucket*). As the output buckets fill up, we write them to disk. As
614 a result, a bucket may be of size up to $P(R)$ (if the data are highly
615 skewed).
- 616 • If we assume “no skew”, operationally this means that each bucket
617 will be of roughly equal size after hashing. We will assume there
618 is no skew. Hence, the number of pages in each bucket under this
619 idealized assumption is $\lceil \frac{P(R)}{B} \rceil$. We make this *uniform hashing or no*
620 *skew assumption below*.

621 We can repeat the partitioning above to further partition the data. In the
622 next phase, we take each input bucket one at a time and treat it as a new
623 input. That is, we partition the data using a new hash function h_2 . Each

624 bucket produces B new buckets, hence The output is a set of B^2 buckets. We
 625 can repeat this process several times. After m passes, this process creates
 626 B^m buckets.

- 627 • Observe that on each pass, we use a distinct hash function.¹² We use
 628 hash function h_i on round i . Note that two tuples s, t are in the same
 629 bucket after m passes if:

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

630 Alternatively, for two tuples s and t if there is even one i such that
 631 $h_i(t[A]) \neq h_i(s[A])$ for $i \in \{1, \dots, m\}$ then, s and t hash to different
 632 buckets.

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

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

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

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

- 646 • Observe that in each pass, we read and write each page in the rela-
 647 tion exactly once. Thus, in m passes, the partitioning phase requires
 648 $2P(R)m$ IOs.
- 649 • The assumption of no skew is likely not met in practice, which is why
 650 we use statistics to estimate the skew (e.g., histograms). This is a
 651 critical point and where our simplified analysis breaks down: without

¹²If we use the same hash function, the bucket size would not reduce.

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

656 • We create a separate function `PartitionTimes(R, A, \bar{h}, m)` in which
657 we replace the termination condition of the while loop of Algorithm 3
658 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
 $\text{Buckets_of_R} \leftarrow \text{Partition}(R, A, \bar{h}, B - 1);$
% Each bucket of R occupies B - 1 pages;
 Suppose $|\text{Buckets_of_R}| = B^m;$
 $\text{Buckets_of_S} \leftarrow \text{PartitionTimes}(S, A, \bar{h}, m);$
foreach *Bucket* B_R **in** Buckets_of_R **do**
 | Let B_S be the corresponding **joinable** bucket to B_R ;
 | $\text{BNLJ}(B_R, B_S, B)$. *% Runs in 1 pass, since B_R fits in memory!*
end

Algorithm 4: Hash Join

659 6.4.2 Join Phase

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

- 665 • First, partition each R and S on the join key A until the buckets
 666 produced by *either* R or S is “small”. Below, we will see that “small”
 667 means $B - 1$ given $B + 1$ buffer pages. Let m be the number of passes.
- 668 • For each of the m passes, we use the same function for R and S . That
 669 is in phase i , we use h_i for *both* relations. This ensures that if two
 670 tuples agree on the join key, then they will be in the *same bucket* after
 671 partitioning.

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

- 674 • The key property of the partitioning is that since we use the same
 675 family of hash functions on each pass, we can select pairs of buckets
 676 B_R from R and B_S from S such that for any tuples $r \in B_R$ and $s \in B_S$,
 677 we have

$$h_i(r[A]) = h_i(s[A]) \text{ for each } i = 1, \dots, m.$$

678 Notice that this holds for exactly one pair of buckets. We call such
 679 pairs of buckets **joinable**. In particular, *if r and s can join, then they*
 680 *are in joinable buckets.* Not all tuples will have the same value of A
 681 in a bucket, so not all tuples in the buckets participate in a join.

682 • For each such pair of joinable buckets, we can run BNLJ. More directly,
 683 we compute B_R joined with B_S . Note that we use our $B + 1$ buffer
 684 pages as follows:

- 685 – $B - 1$ pages for B_R
- 686 – 1 page to hold a single page B_S at a time.
- 687 – 1 page to hold the output.

688 Since we have $|B_R| \leq B - 1$ from the partitioning phase, we hold
 689 the whole block B_R in memory and scan through B_S one page at a
 690 time. We then compute whether each pair of tuples in these blocks
 691 match. In IOs, this costs the number of pages in B_R plus the number
 692 of pages in B_S . Since these are partitions and joinable buckets in a
 693 $1 - 1$ relationship, the total cost of this phase is $P(R) + P(S) + \text{OUT}$.

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

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

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

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

702 6.5 Join Algorithm Comparison

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

710 7 Histograms

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

- 714 • The cost formulas above rely on things like skew- we need to estimate
715 that. If we have more than one join, we need to estimate its output
716 size.
- 717 • You needed to know about ANALYZE. This is probably one of the most
718 practically useful things we teach you in this course: *if your query*
719 *seems slow, make sure it has the best available statistics.*
- 720 • Maintaining accurate statistics seems easy but is fairly tricky: we want
721 to do so in the smallest amount of space we can!
- 722 • Estimation with histograms, this is done in Lecture 17 and you should
723 know how to do it.

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

- 727 • *Suppose one bucket for $R[A]$ spans 5 values with frequency 10, we*
728 *estimate that each value occurs twice ($10/5 = 2$).*
- 729 • *Suppose one bucket for $S[A]$ spans 6 values with frequency 18, we es-*
730 *timate that each value occurs three times ($18/6 = 3$).*
- 731 • *Suppose these buckets overlap on 4 values on A , then we estimate*
732 *$(2 * 3) * 4 = 24$ values produced by this pair of buckets in the join.*

733 To obtain an estimate for the entire output, we sum the contributions of each
 734 pair of buckets to produce the estimate for the join. However, some pairs of
 735 buckets will not overlap at all, and so will contribute 0 to the output.
 736 An alternate strategy, is that we could also write it like this:

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

737 here $\text{Dom}(A)$ is the domain of values for A . Using our standard histogram
 738 algorithm from class, we can estimate $|\sigma_{A=a} R|$ for any value a .
 739 The first algorithm above just uses the buckets, which may be more effi-
 740 cient if the buckets are large (which is not a major concern). The reason to
 741 introduce it is to make sure you know how to operate on the buckets them-
 742 selves.