$[R]$: the # of pages to store R

P_R = # of records per page of R

$|R|$ = # of records of R ($P_R * [R]$)

SNLJ: $[R] + |R|[S]$ — R should be smaller

PNLJ: $[R] + ([R] * [S])$

BNLJ: $[R] + \lceil [R]/(B-2) \rceil * [S]$

INLJ: $|R| *$ (lookup cost) $+ [R]$

lookup = cost to get to leaf +
Alt 1: 0
Alt 2/3 clustered: 1 per matched page of S
Alt 2/3 unclustered: 1 per matched tuple of S

Naive Hash Join

Load all R into hash table
Scan S and probe R
$R < (B-2) *$ hash_fill

Grace Hash Join

Good if input hashed, need output hashed

- Partition tuples from R and S by join key
- Build and probe for each partition

cost: $2 * k * ([R] + [S]) + [R] + [S]$

k = # of passes to get partitions of R to fit in mem

$R < (B-1)(B-2) \Rightarrow k = 1$

Partitioning Phase: divides R into (B-1) runs of size $[R]/(B-1)$

Matching Phase: requires each $R/(B-1) < (B-2)$

Does not include cost of writing matching output

Sort-Merge Join

Requires equality predicate (Equi-joins & natural joins)

Cost: Sort R + Sort S + $([R] + [S])$

last term $([R] * [S])$ worst case (all tups match)

$B > \sqrt{\max([R], [S])} \Rightarrow$ Sorted in 2 passes

$B * (B-1) > \max([R], [S])$

not sensitive to data skew

Good if input sorted, or need output sorted

Parallel

Speed up – Increase HW – Fixed workload → throughput ↑

Scale up – Increase HW, Increase HW → throughput / const

Shared Nothing: runs on commodity HW, scales up w/ data, doesn't rely on HW

Inter-query – Treat each query separately → pipeline, bushy tree

Intra-query – Inter-operator, intra-operator
↳ Partition Parallelism

Data Partitioning

Range, Hash, Round-Robin

Symmetric Hash Joins

1. Each node allocate two hash table, one for each side
2. For each tuple R
   a. build into R for join key
   b. probe into S for matches
3. For each tuple S
   a. symmetric to R

Parallel Sort Merge Joins

one for each relation

Pass 0 ... n-1 one like parallel sorting

Pass n: merge join partitions locally on each node

Parallel Aggregation

$sum(S) = \sum \sum (S)$

$count = \sum count(S)$

$avg(S) = \sum \sum (S) / \sum count(S)$

Asymmetric shuffle

- only partition S since R is already partitioned properly
- If R partitioned, just partition S, then run local join at every node and union the result.

Broadcast Joins

- R is small, send R to every partition of S
- Do local join then union

Parallel sort

Pass 0: shuffle data across machines

Pass 1-n: done independently, single-node sorting

Pipeline Breaker

Sort, so can't start until sorted

Hash build, though hash can't start probing until hash table built

Query Parser: checks correctness, auth, generates a parse tree

Query rewriter: converts queries to canonical form (flatten views, subqueries into fewer blocks)

Query Opt

Plan space, cost estimation, search strategy

Heuristics

- selection cascade and pushdown
- projection " " "
- avoid cartesian products

Physical Equivalences

- Base table access – Heap scan, Index scan
- Equi-joins – Block, Index Nested, SM Join, GM Join
- Non-Equi → block Nested Loop

System R

Cost: # I/Os + CPU factor * # tuples

Catalog

NTuples	# of tuples in table
NPages	# of disk pages in a table
Low/High	min/max value in a col
Nkeys	# of distinct in a col
IHeight	height of an index
INPages	# of disk pages in an index

Result size est = max # tuples * product of all selectivities

Term col = value (given Nkeys(I) on col)
sel = 1/Nkeys(I)

Term col1 = col2 (handy for joins)
sel = 1/Max(Nkeys(I1), Nkeys(I2))
↑ needs index on both otherwise 1/Nkeys(I)

Term col > value
sel = (high(I) − value) / (high(I) − low(I) + 1)

If missing → 1/10

Subtract ∩ for OR

Alternative Plans

Single table
↳ selects, projects, groupBy/Agg
↳ fileScan / index

Cost Estimates for Single Plan

Index I on primary key matches selection:
cost = (Height(I)+1) + 1
for B+ Tree

Clustered Index I matching selection:
(NPages(I) + NPages(R)) * selectivity

Non-clustered Index I matching selection:
(NPages(I) + NTuples(R)) * selectivity

Sequential scan of file: NPages(R)

Recall: Must charge for dup elimination if required

Left-deep plans differ in
- order of relations
- access method for each leaf operator
- the join method for each join operator

Enumerated using N passes
Pass 1: Find best 1-relation plan for each relation
Pass i: Find best way to join result of an (i-1)-relation plan to the i-th relation. (i between 2 and N)
Keep cheapest plan overall and each interesting order

Interesting Orders
Index scans outputs sorted on index column
Sorted by anything that we can use later
↳ ORDER BY / GROUP BY
↳ Join attributes of yet-to-be-added joins

IR's Bag of Words
Given corpus of text file, build table
InvertedFile (term text, docID text)
keep posting list like Alt 3 sorted by DocID
near → add position to inverted file and posting list, check that positions are k off

Hellerstein Inequality
$\frac{d\,app}{dt} << \frac{d\,env}{dt}$ Data indep. is most important when this holds.

Logical data indep.
Maintain views when logical struct changes

Physical data indep.
Maintain logical struct when physical struct changes

Data model = collection of concepts for describing data

Schema = description of a particular collection of data, using a given data model

Relational model of data
- main concept = relation, rows and cols
- every relation has a schema

2PC Recovery
Coordinator recovers
- w/ commit and end → nothing
- w/ commit only → rerun phase 2
- w/ "abort" → nothing

Participant recovers:
- no prepare/commit/abort? Nothing
- w/ prepare & commit? send Ack to coord
- w/ just prepare? send inquiry to coordinator
- w/ "abort" Nothing

Commit iff coordinator logged a commit

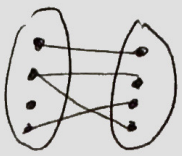
Selectivity Est
$c >= V \Rightarrow \frac{(high\text{-}key - V)}{(high\text{-}key - low\text{-}key + 1)} + \frac{1}{\#\text{ of distinct vals of } c}$

$c < V \Rightarrow \frac{(V - low\text{-}key)}{high - low + 1}$

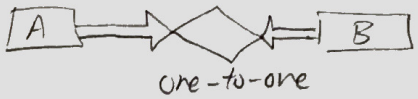
$c <= V \Rightarrow \frac{(V - low\text{-}key)}{(high - low + 1)} + \frac{1}{\#\text{ of distinct values of } c}$

Inequalities & Float
$c >= V \Rightarrow \frac{(high - V)}{high - low}$

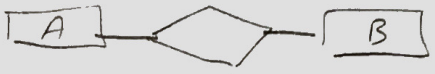
$c <= V \Rightarrow \frac{(V - low)}{high - low}$



1-to-many | many-to-1



one-to-one



many-to-many

1-to-many

ER
Entity - Real world obj described by attrs
Entity set - collection of the same type of entities
Each entity set has a key
Each entity has a domain

Relationships - Association among 2+ entities
relationship set - collection of relationships involving the same entity sets
↳ An n-ary relationship set R relates n entity sets $E_1 \ldots E_n$ each relationship in R involves $e \in E_1, \ldots, e_n \in E_n$

key constraint (at most one) →

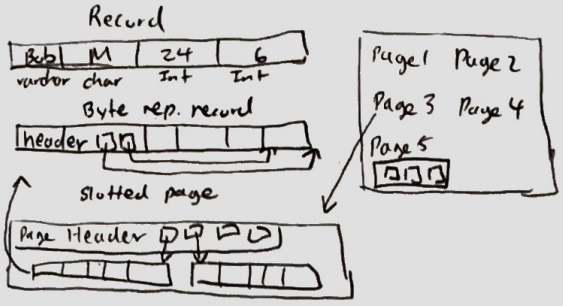
participation constraint (at least one) ≈≈≈

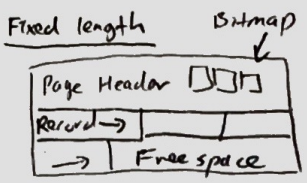
key constraint w/ total participation (exactly one) ⇒

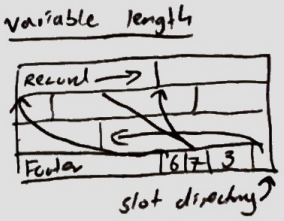
Non-key partial participation (0 or more) —

Weak Entities
- Identified uniquely only by considering primary key of another entity
  ↳ owner set and weak entity set must be one-to-many (owner) (weak entities)
  ↳ weak identity must have total participation in this identifying relationship set

SQL client
Query Parse & Opt
Relational Op
Files and Index Manage
Buffer Manage ← RAM ← DISK
Disk space Manage
Database







Packed → rearrange
Unpacked → clear bit to delete

Indexed Sequential Access Method
affects part of the index

Cost Model

B: # of data blocks in file
R: # of records per block
D: Avg time to R/W disk block
F: Avg internal node fanout
E: Avg # data entries per leaf

	Heap	Sorted	Clustered Index
Scan	$B*D$	$B*D$	$3/2 * B * D$
Eq	$0.5 * B * D$	$\log_2 B * D$	$(\log_F(BR/E) + 2) * D$
Range	$B * D$	$(\log_2 B + \text{pages}) D$	$(\log_F(BR/E) + \# + 3 * \text{pages}) * D$
Insert	$2 * D$	$(\log_2 B + B) * D$	$(\log_F(BR/E) + 4) * D$
Delete	$(0.5 * B + 1) * D$	$(\log_2 B + B) * D$	$(\log_F(BR/E) + 4) * D$

B+ Tree

$F = 2d + 1$

$d \leq$ # entries $\leq 2d$

Records $= F^h \cdot 2d$

bulk load
↳ sort first then insert up to fill factor

leaf split into leaves with $(d, d+1)$ entries and copy up
inner split into nodes with $(d, d+1)$ entries and push split key up

	Heap	Sorted	Clustered Index
Scan	$O(B)$	$O(B)$	$O(B)$
Eq	$O(B)$	$O(\log_2 B)$	$O(\log_F B)$
Range	$O(B)$	$O(\log_2 B)$	"
Insert	$O(1)$	$O(B)$	"
Delete	$O(B)$	$O(B)$	"

Terminology

- DB: sets of named relations
- Relation (Table):
  - Scheme (metadata)
  - Instance (set of data satisfying scheme)
- Attr (column, field)
- Tuple (Record, row)

SQL

SELECT [DISTINCT]
FROM
WHERE
GROUP BY
Having
ORDER BY
LIMIT

Can't use aggr without group by

Create VIEW view-name As select_statement (not stored)

WITH name AS
SELECT

Regex

WHERE S.name ~'B.*'

Old school SQL

WHERE S.name LIKE 'B_%'

Sets

Union all: sum of cardinalities
Intersect all: min of cardinalities
Except all: diff of cardinalities

Comparison ops
IN, EXISTS
NOT IN, NOT EXISTS
ANY, ALL

Use IS or IS NOT for null

Replacement Policy

LRU → costly, works well for repeated access to popular pages
Sequential flooding
MRU → $\sim(B-1)/N$ hit rate for sequential scan

Data Entry Storage

affects part of index

Alt 1: by value
Alt 2: by Ref
Alt 3: by list of refs

2, 3 can be clustered or unclustered

clustered: efficient for range searches, potential locality, supports compression, more $ to maintain

Search key

A composite search key matches a query if:
query is a conjunction of $m \geq 0$ equality clauses of the form: $k_1 = \langle val_1 \rangle$ and $k_2 = \langle val_2 \rangle$ AND ... $k_m = \langle val_m \rangle$ and at most 1 additional range clause of the form: AND k_{m+1} op $\langle val \rangle$ where op is one of $\{<, >\}$

choice of search key
↳ affects the queries for which we can use an index

External sorting (Conquer & Merge)

of passes $\lceil \log_{B-1} \lceil N/B \rceil \rceil + 1$
cost $= 2N *$ # of passes
Two passes → $B(B-1)$ pages
sort N pages in about $B = \sqrt{N}$ space

Hashing (Divide & conquer)

Two passes → $B(B-1)$
Hash $(B-1)$ partitions of length $\leq B$
$> B$ requires recursive partitioning
cost $= 2N *$ # of passes
good for removing duplicates

Relational Algebra

- set semantics

projection (π): retains columns
selection (σ): subset of rows
renaming (ρ): rename attr and relation
Inter ($\cap$): tuples in r1 and r2
Joins ($\bowtie_\theta$, $\bowtie$): combine relations that satisfy predicate
Union ('$\cup$): must be compatible ↳ matching fields
set diff ($-$): SQL Except
cross-prod ($\times$): each row in r1 paired w/ each row in R2

Group By (γ): $\gamma_{age, AVG(rating), Count(*) > 2}(sailors)$
Agg op

Cost of External IMS

B=5 N=108

Pass 0: ⌈108/5⌉ = 22 sorted runs of 5 pages each

Pass 1: ⌈22/4⌉ = 6 sorted runs of 20 pages each

Pass 2: ⌈6/4⌉ = 2 sorted runs of 80 and 28 pages

Pass 3: sorted 108 pages

Classical IR Ranking: vector space model, similarity is the cos of the angle between two normalized vectors or dot product

TF x IDF (Term Freq x Inverse Doc Freq) = DocTermRank

occurrence counts aren't good term weights, want to measure repeated or unusual word, IDF = log(total # docs / #docs w/ term)

Precision = TP/Selected Recall = TP/(TP+FN)

FDs

Superkey: set of columns that determines all cols in table

Candidate key: a minimal set of columns that determines all cols

$K \rightarrow \{\text{all attributes}\}$, for any $L \subset K$, $L \not\rightarrow \{\text{all attributes}\}$

Update anomaly: need to update all records w/ same key

Insertion anomaly: need to insert but don't know the val?

Deletion anomaly: If we delete all record w/ a value, we lose info

F+ = closure of F: the set of all FDs implied by F (includes trivial dependency)

Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$ Augmentation: If $X \rightarrow Y$, $XZ \rightarrow YZ$

Transitivity: $X \rightarrow Y$, $Y \rightarrow Z$, then $X \rightarrow Z$ Union $X \rightarrow Y$, $X \rightarrow Z$, then $X \rightarrow YZ$ Decomp: $X \rightarrow YZ$, then $X \rightarrow Y$, $X \rightarrow Z$

Find closure:

$X+ := X$

Repeat until no change:

for $U \rightarrow V \in F$,

if $U \subseteq X+$, then add V to X+

If you can guess missing val, then not BCNF

Loss less if joining decomposed relations gets back original

decomp of R into X and Y is lossless w.r.t. F iff the closure of F contains $X \cap Y \rightarrow X$ or $X \cap Y \rightarrow Y$

Decompose R into X and Y is dep preserving if $(F_X \cup F_Y)^+ = F^+$

Insert/Delete

↳ 1 to read record

1 to write record

1 to update leaf node

Eq Search

↳ 1 to read the leaf

↳ add 1 to read actual records

Range Search

↳ $\frac{3}{2}$ to read records > 3+ pages

↳ $\frac{3}{2}$ to read leaf

(2/3 Fill Factor)

If BCNF → 3NF but not other way around

SQL Joins

```
SELECT H.home_ids, T.sale_price
From Homes H
LEFT OUTER JOIN Transactions T
ON H.home_id = T.home_id
WHERE H.city = "Berkeley"
```

```
SELECT T1.did
FROM Trip AS T1
WHERE 5.0 = ALL (
    SELECT T2.driver_rating
    FROM Trip AS T2
    WHERE T1.did = T2.did AND DriverRating IS NOT NULL)
```

May not be a dep preserving decomp into BCNF

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	
IX	✓	✓	.		
S	✓		✓		
SIX	✓				
X					

Sort-Merge Refinement

pass 0 - k-2 external sort R and S until almost sorted

pass k-1 merge R and S

$2*(k-1)*([R]+[S]) + [R]+[S]$

2 pass requires $B \geq \sqrt{R} + \sqrt{S}$

LogRec: LSN, prevLSN

DB: db pages each w/ pageLSN

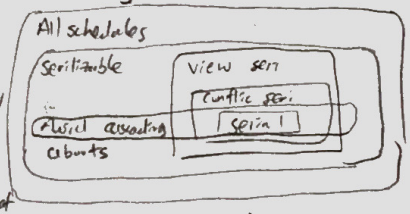
RAM: Xact Table (xid, lastLSN); DPT (recLSN); LogTail (flushedLSN)

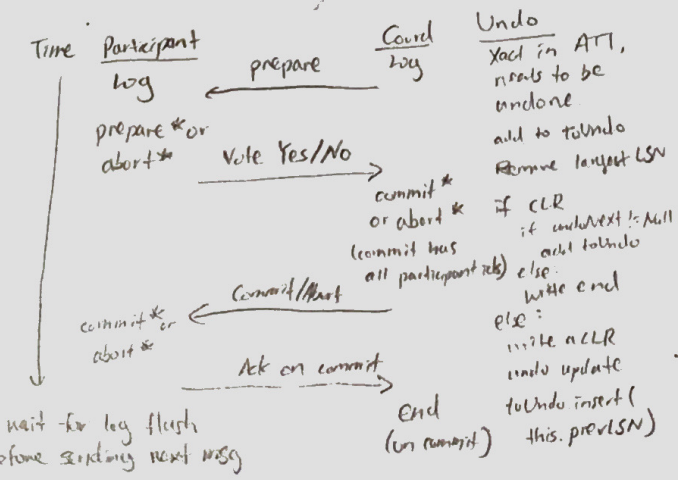
Atomicity = All or None

Consistency: DB starts and ends xact consistent

Isolation = Exec of each xact is isolated

Durability: if xact commits, persist





Undo

Xact in ATT, needs to be undone

add to toUndo

Remove largest LSN

if CLR

if undoNext != Null add toUndo

else write end

else:

write a CLR

undo update

toUndo insert (this.prevLSN)

Analysis

Scan log from chkpt

- End → Remove from Xact Table

Update/CLR

If not in ATT, add

else update lastLSN

If not in DPT, add

Abort/Commit

Update ATT (status, lastLSN)

Redo

Update/CLR

Redo unless any of this is true

- Affected page not in DPT
- Affected is in DPT, but recLSN > LSN
- PageLSN (in DB) ≥ LSN

$T_i > T_j$? wait : die

$T_i > T_j$? wound : wait

No Steal Steal

	No Undo	UNDO
No Force	No Undo Redo	UNDO REDO
Force	No Undo No Redo	UNDO No Redo