CS186 MT2

ITERATORS & JOINS:

hasNext(): Is there more data?
next(): Fetch the next peice of data
[R = pgs * P/pgs]
cardinality, total no records |R|, data pgs.
= total no of pgs * no of records/pg
[R] Pr

DIFFERENT JOINS

compute both ways for R⋈S & check which one gives smaller cost.

SMJ: [Sort Merge Join]

cost: cost of sorting R + cost of sorting S + cost of merging
cost of sorting a relation w N pgs: $2N * ceil(1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$
cost of Merging:
WC: [R] + |R| * S Avg C: [R] + [S]

GHJ:

COST: cost to hash [R] & [S] + cost of Native Hash Join
hash until ≤ B-2
write: ceil(no of pgs to write / B-1)
total I/Os: ceil(no of pgs to write / B-1) * B-1 + Read I/Os
Native hash join: [R] + [S] I/Os - (last passes I/O write)

BLNJ: min(R⋈S, S⋈R)

COST: [R] + ceil([R] / (B-2)) * [S]

SNLJ: min(R⋈S, S⋈R)

COST: [R] + P_R[R][S]

PNLJ:

keep smaller relation b/w R + S as the outside loop
COST: [R] + [R][S]
* need at least 3 buffer pgs

INLJ: [Index Nested]

[R + W + join]
COST: [R] * |R| * (cost to lookup matching record)
Clustered: 1 I/O / pg of matching record
Unclustered: 1 I/O / matching record.

NOTE:

INLJ, SMJ joins based on data being & GHJ - ordered a certain way (equality predicate)
BLNJ, SNLJ, PNLJ - handles all conditions.

Alt 1 has I/O cost of => default sorted.
(height)
cost to reach above leaf + num of leaves read.
Alt 2/3 has I/O cost of
cost to reach above leaf + num of leaves read + num data pages read.
cost to reach above leaf is generally Height of tree, h,
but when given matching tuples its h+1.

EXAMPLE: Grace Hash Join:

Grace Hash Join:
Justice League = 100 pgs ½ of records = earth: — equality predicate.
T = 200 pgs
B = 12

J
pass 1 read 100 pgs = 100 I/Os
pass 1 write: ceil(no of pgs to write / B-1) since only half
= ceil(50/11) = 5
∴ total I/Os: 5 x B-1 = 5x11 = 55

T
200 pgs = 200 I/Os
ceil(200/11) = 19
total I/Os: 19 x 11
= 209

∴ total cost: 100 + 55 + 200 + 209 + 55 + 209 = 828 I/Os

QUERY OPTMIZATION:

Given a plan some things we can do to cheapen a query:
① push selections / projections down a tree
② Materialize immediate relations (write to a temp file)
→ results in additional write I/Os but better in the long run
③ Use indices INLJ.

SELECTIVITY: % of tuples selected by a predicate.
STEPS: selectivity * # tuples * if not stated use 1/10

EQUALITIES:

Predicate	Selectivity	Assumption
c = v	1 / (number of distinct values of c in index)	We know |c|.
c = v	1 / 10	We don't know |c|.
c1 = c2	1 / MAX(number of distinct values of c1, number of distinct values of c2)	We know |c1| and |c2|.
c1 = c2	1 / (number of distinct values of ci)	We know |ci| but not |other column|.
c1 = c2	1 / 10	We don't know |c1| or |c2|.

INEQUALITIES OF INTEGERS:

Predicate	Selectivity	Assumption
c < v c > v	(v - min(c)) / (max(c) - min(c) + 1) (max(c) - v) / (max(c) - min(c) + 1)	We know max(c) and min(c). c is an integer.
c < v c > v	1 / 10	We don't know max(c) and min(c). c is an integer.
c <= v c >= v	(v - min(c)) / (max(c) - min(c) + 1) + (1 / |c|) (max(c) - v) / (max(c) - min(c) + 1) + (1 / |c|)	We know max(c) and min(c). c is an integer.
c <= v c >= v	1 / 10	We don't know max(c) and min(c). c is an integer.

INEQUALITIES OF FLOATS:

Predicate	Selectivity	Assumption
c >= v	(max(c) - v) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c >= v	1 / 10	We don't know max(c) and min(c). c is a float.
c <= v	(v - min(c)) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c <= v	1 / 10	We don't know max(c) and min(c). c is a float.

CONNECTIVES:

Predicate	Selectivity	Assumption
p1 **AND** p2	S(p1)*S(p2)	Independent predicates
p1 **OR** p2	S(p1) + S(p2) - S(p1)*S(p2)	Independent predicates
NOT p	1 - S(p)	

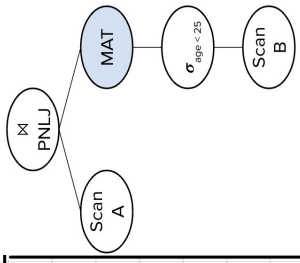
2 cases: if predicates dependent on each other.

P(A∩B) = P(A) + P(B) - P(A∩B) - dependent.
P(A∪B) = P(A) + P(B) - total exclusivity

EXAMPLES:

Select * FROM R INNER JOIN S on R.col = S.col WHERE R.col < 5
B=12 [R] = 20 pgs [S] = 10pgs Full scan for R: 20 IOs & 5 pgs of o/p
Whats the cost of R BNLJ S? Full scan for S: 10 I/Os
Ans: 20 + ceil(5/12-2) * 10 = 20 + 1*10 = 30 I/Os
but ∵ we push down only 5 is used.

Table A: takes 50 I/Os to perform a scan
Table B: takes 100 I/Os to perform a scan
Sel(B.age < 25) = 0.5, [B] = 100
By materializing the intermediate relation, we're applying $\sigma_{age<25}$ before PNLJ, and performing the join on the *result* of the selection.
Cost = *Scan A* (50) + *Scan B* (100)
+ *Materialize* (100 * 0.5) + *PNLJ* (50 * 50)
→ 2,700 I/Os in total



TRANSACTIONS & CONCURRENCY

A: Atomicity: All ops OR NO ops [commit or abort]
C: Consistency: Data is constant & maintained
I: Isolation: 1 transaction at a time
D: Durability: transaction persists after committing

Throughput: no of transactions / unit time
Latency: response time / transaction

Serial Schedule: each transaction runs from start to finish w/o intervening actions from other transactions.
We use serializability to provide isolation

Check for Serializability.
- same intial reads - starts the same
- same winning writes - ends the same

2 schedules in conflict:
- diff transac^n
- same resource
- 1 write at least

conflict equivalent:
- ops within each transaction is same b/w both schedules
- every pair of conflict ops are ordered same way.

conflict serializable:
S is conflict = some serial schedule
⇒ S is serializable.
- no cycles in dependency graph.

view serializable:
- same intial read
- same fpdt reads
- same winning writes.

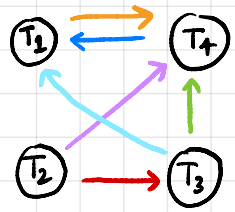
NOTE:
Blind Writes: writes w no Rs b/w them.

conflict serializables care abt order of BW!

T1					R(A)		W(C)		R(B)
T2		R(A)	R(B)						
T3	R(A)					W(B)			
T4				W(A)				R(C)	

Reads + writes go together.

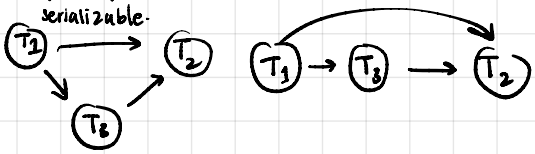
i) R(A) before W(A)
ii) R(A) before W(A)
iii) W(A) before R(A)
iv) R(B) before W(B)
v) W(C) before R(C)
vi) W(B) before R(B)



- cycle in graph so not conflict serializable.

T1	R(A)						R(C)	W(B)	COMMIT		
T2				R(A)	W(A)	COMMIT					
T3		R(A)	R(C)							W(B)	COMMIT

yes conflict serializable.



SIMPLE LOCKING:

S: lets transaction read (R) [Shared]
- many transacn hold S at once

X: lets transaction modify (W) [Exclusive]

	NL	S	X
NL	✓	✓	✓
S	✓	✓	X
X	✓	X	X

DEADLOCK: $T_1 \xleftrightarrow{\text{WAITS}} T_2$

Waits-for graph
if Ti holds lock that conflicts w lock Tj wants
"Tj waits for Ti" $(T_j \longrightarrow T_i)$

PRIORITY: current time - start time
(older ↑ ∝ Priority ↑)

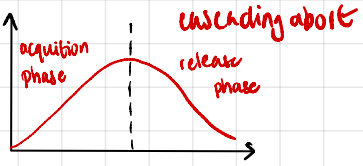
high low
wait-die: Ti wants a lock but Tj holds conflict
Ti high? waits for Tj
Ti low? aborts (die)

high wound - low wait: Ti wants a lock but Tj holds conflict
Ti high? ABORT Tj (wounds)
Ti low? waits for Tj

2 PHASE LOCKING (2PL)

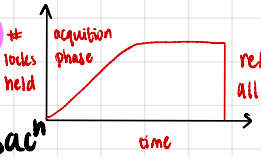
problem

PROBLEM:
2PL lets new transacn read new vals b4 tracn COMMITS



STRICT 2PL:

- allows release at end of transacn acquire b4 abort/commit.



Consider the following schedule:

T1	R(A)						R(C)	W(B)	COMMIT		
T2				R(A)	W(A)	COMMIT					
T3		R(A)	R(C)							W(B)	COMMIT

Which of the following changes to the schedule above will result in a schedule that is possible using strict 2PL?

A) Make T1 abort instead of commit
B) Make T2 abort instead of commit
C) Remove R(A) from T1 and T3
D) Remove T2 from the schedule
E) Swap W(B) with W(D) for T3
F) No change needed, it is already strict 2PL
G) None of these changes provide strict 2PL

A) Incorrect, strict 2PL deals with acquisition of locks **before** commit/abort
B) Incorrect, same as A
C) Correct, T2 can acquire exclusive lock on A before T1 and T3 commit.
D) Correct, Removing T2 means no transaction needs exclusive lock on A
E) Incorrect, T2 still can't acquire W(A) until T1, T3 commit
F) Incorrect, see E.
G) Incorrect

Consider the following schedule:

T1	R(A)	W(A)		R(B)	W(B)		W(A)	COM					
T2			R(B)								COM		
T3						R(C)				W(C)		COM	

Does the schedule follow 2PL?

Yes, no transaction acquires a lock after releasing a lock. Consider T2: it needs to release S(B) before T1 can acquire X(B), but T2 does not acquire any more locks after this.

Does the schedule follow strict 2PL?

No, under strict 2PL, T2 would not release its locks until it is ready to commit. This means T1 should be unable to acquire X(B) until T2 commits.

MULTI-GRANULARITY LOCKING:

Compatibility matrix:

	NL	IS	IX	SIX	S	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	X
IX	✓	✓	✓	X	X	X
SIX	✓	✓	X	X	X	X
S	✓	✓	X	X	✓	X
X	✓	X	X	X	X	X

Parent-Child lock relationship:

Parent	Pass Child locks
NL	None
S	None
X	None
IS	IS, S
IX	IX, X, IS, S
SIX	IX, X

* IS: Intent to acquire Shared lock at a lower level.
IX: Intent to acquire eXclusive lock at a lower level
SIX: Shared Intent to acquire eXclusive lock at a lower level
→ can read entire table + acquire X lock
→ prevents writes but allows reads!

We have a table, T, with pages A and B. Page A has tuples A1, A2 and Page B has tuples B1, B2
1) List the locks T1 has at the end of timestep 5 in the order they were acquired.
2) What locks are on T2's waiting queue at the end of timestep 5?

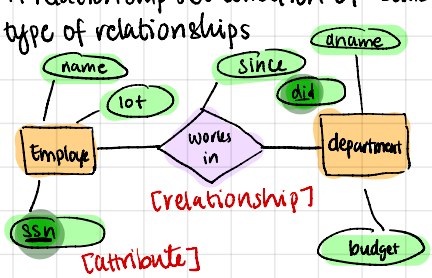
time	1	2	3	4	5
T1	R(A1)		R(A)		W(A)
T2		W(B1)		W(A2)	

1)
After timestep 1: T1 has IS(T), IS(A), S(A1)
After timestep 3: T1 has IS(T), S(A) (T1 escalates on the page-level lock for A)
After timestep 5: T1 has **IX(T), X(A)** (T1 promotes S(A) to X(A) - note: promote does not affect acquisition order)
2)
T2's waiting queue: acquire IX(A)
Note that T2 will want to acquire X(A2) eventually, but it gets blocked on trying to acquire IX(A) before it gets the chance to try to acquire X(A2).

ER DIAGRAMS:

entities are real world obj described with attributes.

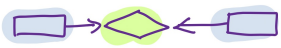
A relationship set collection of same type of relationships



many-to-many

many-to-one:

'onal partic.
one-to-one



Constraints:
→ : at most one (key const)
— : at least one (participation const)
⇒ : exactly one (key + total participation)
—— : 0 or more (non-key partial participation)

weak entities: can be identified uniquely only w/ the key of another entity.
↳ partial key (---) identifier + owner entity's key when combined w/ entity's key
↳ must be 1-to-many rel. w/ total partic.

Inference Rules:
$Y \subseteq X \Rightarrow X \to Y$
$X \to Y \Rightarrow XZ \to YZ$
$X \to YZ \Rightarrow X \to Y$ and $X \to Z$
$X \to Y$ and $Y \to Z \Rightarrow X \to Z$
$X \to Y$ and $X \to Z \Rightarrow X \to YZ$

* BCNF always lossless.
BCNF (Boyce-Codd Normal Form): ← no redundancy.
R in BCNF frm if for every FD $X \to A$ that holds over R, either ① $A \subseteq X$ or ② X is a superkey.
algo: for each FD $X \to Y$ in F^+:
if $X \to Y$ violates BCNF → decomp R into $(R - X^+) \cup X$ and X^+.
* don't want lossiness!
Lossless iff F^+ contains
ⓐ $(X \cap Y) \to Y$ or ⓑ $(X \cap Y) \to Y$.

FUNCTIONAL DEPENDENCIES

functional dependency
$X \to Y$ (X determines Y)
superkey: X is superkey of R if $X \to$ all attributes of R
candidate key: a set of keys that determines all columns in a relation
attribute closure: of an attribute X given a set of FDs is X^+
Relation R is BCNF if:
for every FD $X \to A$ that holds over R either $A \subseteq X$ or X is a superkey.
- no redundancy in BCNF!
lossiness - NOT able to construct original relation.
never lossless.

Closure: $F = \{A \to B, AB \to AC, BC \to BD, DA \to C\}$
A^+: $A \to B$ (AB)
$AB \to AC$ (ABC)
$BC \to BD$ (ABCD)
BC^+: $BC \to BD$ (BCD)
AB^+, AC^+, AD^+ = ABCD ⇒ superkeys b/c AD candidate key.
A → candidate key b/c A^+ = minimal set of keys to cover all symbols in F.

RECOVERY:

ideal 4 I/O performance

NO STEAL:
- if pgs not flushed to disk yet trxn cannot steal
- SLOW - other trxn wait for pgs.
- few uncommitted transactns

STEAL:
- evict/flush pgs w uncommited updates
- FAST but on crash, need to UNDO unwanted updates (for Atomicity)

NO FORCE:
- commit w/o forcing dirty pg to disk
- FAST but on crash need to REDO for durability.
- guaranteed during buffer manager's MRU pg replacement policy.
- less I/Os, Durability

FORCE:
- force dirty pgs to disk on COMMIT.
- SLOW: I/Os to write pg to disk

Atomicity

	No steal	Steal
No F		FASTEST
F	SLOWEST	

Performance

	No Steal	Steal
	no undo redo	undo redo
F	no undo no redo	undo no redo

Logging / Recovery

Steal, No Force: Write Ahead Logging

- log records written to disk before data pg to disk.
- all log records written to dis when transacn commits.

Undo Logging: we want to Undo if not committed.

4 types: start commit abort update.

1) if T modifies element X, update log record written to disk before dirty page. [STEAL]

2) if T committed, WRITE to disk before commit scan log from end to find if T completed. [FORCE]

if T not completed write X = v to disk.

Redo Logging: no force, no steal

redo all Ts, update + commit record written before dirty page.

ARIES Recovery Algo: Analysis Redo, Undo.

Analysis: rebuild the trxn table, Dirty Page table (DPT)

if not END, add to trxn table, set last log sequence number. (LSN)

if COMMIT or ABORT, transcn status change

if UPDATE, not in DPT, add to DPT, recLSN to LSN.

if END, remove from transacn table.

ARIES: Recovery system → Steal, no force
- uncommitted data can be flushed (↑ mem)
- data pgs don't need be flushed b4 xact can commit (↓ latency of xact).

The Log: (WAL) → Log recs are assigned an LSN (log seq. #); prevLSN - LSN of prev log rec for xact. Compensation Log Rec (CLR) log entry undoing another non-CLR log entry.

Memory	Log	DB
• Xact Table (last LSN, status) • DPT → recLSN • flushed LSN • Buffer Pool • Log Tail	• LSN • prevLSN • XID • type • pg ID • len • offset • bA-img • after-img	• Data pgs (+ ea w/ pgLSN) • Master Rec.

Memory:
- Xact Table → tracks active xactions.
- DPT: tracks dirty pgs in buffer.
- recLSN - recovery LSN indicates first log rec that caused pg to be dirty.
- flushed LSN: max LSN successfully flushed

Log Recs: update, commit, abort, end, CLR, begin checkpoint, end checkpoint.
* ea rec associated w/ a txn.

Pages: pgLSN = LSN of last rec modifying a pg. stored on pg itself, flushed w/ pg, can differ in mem v. on disk pg.

ARIES: Commit: ① write commit rec to log ② flush log up to commit rec ③ write end rec.

ARIES: ABORT: ① write abort rec to log ② undo all changes made by txn. (start from lastLSN, go back via prevLSN) (undo all log rec that can be undone by adding CLRs to log) ③ write end rec.

ARIES: Recovery: → analysis, redo, undo →

Step 1: Analysis - start from begin_chkpnt.
↳ update txn table + DPT. → end all committed txns and abort all running txns. * smallest LSN for ea. dirty pg.

Step 2: Redo - start from smallest/oldest recLSN (in DPT) → redo all update and CLR recs UNLESS:
a) pg not in DPT ⇒ disk up to date already
b) recLSN of pg > LSN ⇒ no need to redo, change was already flushed.
c) pgLSN ≥ LSN ⇒ pgLSN is authority to dot flush

Step 3: Undo - start from largest/most recent lastLSN (in txn table) → undo all recs of aborting txns, skipping over CLRs → write CLRs as do undos. Abort all txns in xact table not already committing.

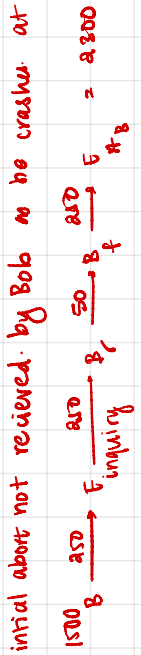
ARIES Checkpointing: so don't need to start from beginning every time!
↳ fuzzy checkpoints → taken while other txns are running + making changes.
→ write a Begin Chkpnt rec (latest rec can start analysis from); write end chkpnt rec later (contains DPT/Xact table valid @ the time of begin checkpoint).

NOTES:

→ 2PC cannot handle indefinite failures
if coordinator doesn't recover no participant gets msg

→ By partitioning/centralizing locks database can still use 2PL

- 2PC w presumed abort does not result in fewer records.



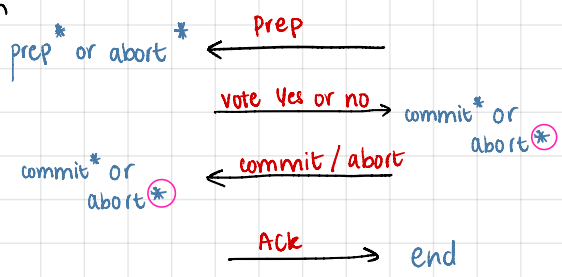
DISTRIBUTED TRANSACTIONS

2PC: 2PHASE COMMIT

2PC: Handling Failures

participants | coordinator

- coord thinks participants went down
  - ↳ if p didn't vote ABORT
  - ↳ if waiting for Ack, RECOVER
- p thinks c went down:
  - ↳ if prep not logged, vote NO
  - ↳ if prep logged, RECOVER



* = flush to disk • optimized (if commit)

2PC Recovery:

- if have commit/abort log rec: do that action
- if have prep log rec, but no commit/abort → send log INQUIRING about status
- if no prep, commit, abort → abort

Optimized: assume can abort if vote NO, don't need ack, abort not flushed, abort as soon as abort is known

NO SQL STUFF

online transacn processing (OLTP): simple lookups
online analytical processing (OLAP): read only queries w/ lots of joins, agg

Scaling: ① **Partitioning (Sharding)** → split data among mult mach. To ↑ parallelism [faster W, slower R]
② **Replication** → copy data among mult mach for fault tolerance (slower writes, faster reads)

Distributed Systems Desired Properties: ① **Consistency:** 2 clients same view ② **availability** every req must get a response ③ **partition tolerance** even if mach disconnects delay or drop packets etc.

CAP Theorem: only can get 2/3 properties (① v ②)

BASE Semantics: eventual consistency systems gaurantee:
① Basic Avail → R/W available ASAP but not always consistent
② Soft State → DB can change even w/o inputs.
③ Eventual Consist → given enuf time all reads consistent after updates propogates.

NO SQL Data Models:
- Data model: (key, val) pairs · ops: get(k), put(k, v)
- Distr / Partit (w/ hash func) → no replicate / multi-way rep → propogate update w/ eventual consistency.

Document Stores: (like JSON) → store collec of docs (data)
JSON: * object (key : value); arrays []; atomics support
* self-describing → schema elems in data itself.

	JSON ← enf on read	Relation ← enf on write.
flexibility	flexible, can rep complex + nested data	Less flexible
schema enf.	self-describing	schema fixed
rep	text-based	B in rep (good 4 disk)

(ex) {"student1": [{"sid": 0001, "name": "me", "classes": [61A, 61B]}, {"sid": 0002...}

MongoDB: store field : value.
· ea. doc has special "_id" as primary key.
↳ syntax on last pg!

Distributed File System (DFS): for v large file; ea chunk replicated several times ≥ 3 for fault total.

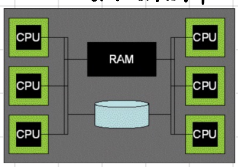
MAP Reduce: high-level prog model & important for large scale parallel processing.
① Map → extract s/t you care abt from each record
② Shuffle & sort
③ Reduce → agg, summarize, filter, transform
④ write results.
* MR slower if complex + need many MRs.

MAP: user provides MAP funct w/ input: (in key, val) and output: bag of (intermediate key, val)
* sys applies map func in parallel to all (in key, val) in the input file
* fault tol by writing intermed files to disk.

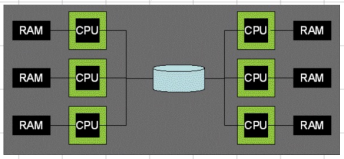
REDUCE: sys groups all pairs w/ same intermed key & passes bag of vals to reduce func user provides reduce func input (interm key, bag of vals);
o/p: bag of o/p vals.

PARALLEL QEURY PROCESSING

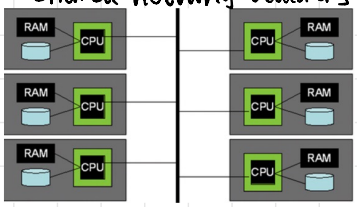
[modern computer] Shared memory



[network file system] shared Disk



Shared nothing [cluster]



Intraquery Parallelism: spread work of one query to multiple machines.

Interquery Parallelism
↳ Intraoperator: make one operator run as (ex sorting multiple) quickly as possible.
↳ Interoperator: running operators in parallel.
↳ Pipeline parallelism: records passed to parent as soon as done
↳ Bushy tree parallelism: diff branch of tree run in parallel.

Interquery Parallelism: gives each one diff qeuries for ↑ throughput & finish more qeuries

Sharding: each data pg stored on 1 machine
Replication: each data pg on multiple machine

Network Cost: how much data to send over network for op

Parallel sorting / hashing: range partition table, local sort/hash on each machine.

Passes: 1 (partition across machines) + ceil[1 + $\log_{B-1}$[N/mB]]
[no of passes to sort table]

SMJ Passes: 2 (1 pass/table to partition across machine) + ceil[1 + $\log_{B-1}$[S/mB]] [passes to sort R+S] + 2 [final merge sort pass]

Partitioning Data:
Range: key ÷'s data based on which range key belongs to.
pro: get(key); sorting; SMJ, etc...
con: distr of data may be screwed.
Hash: ÷ data based on hash func.
pro: get(key); hashing; GHJ, etc...
con: distr may be screwed.
Round Robin: cycles thru partitions as data comes in.
pro: even distr; summing all data across mach.
con: not ordered based on key.

Parallel Sorting:
- partit data over mach w/ range partit.
- perform ext. sort on ea. mach. indep.

Parallel SMJ:
- partit data for both rel over mach w/ range partit. (same range for ea. rel).
- perform SMJ on ea. mach. indep.

Parallel Hashing:
- use hash func to partition data over mach. then, run ext hashing on ea. mach. indep

Parallel Hash Join:
- use hash partit on both rel, then perf a normal GHJ on ea. mach. indep.

Symmetric Hash Joins: → streaming hash join algo → doesn't req all tuples of 1 rel to be available b4 starting.

Broadcast Join:
if R is small, send to ALL nodes that has partition S.
Do a local join at each node (any alg) & union results.

ASSYMETRIC SHUFFLE
if R is partitioned well just partition S then local join at every node & union results.