Today:

- Optimizer architecture: pipelining.
- Resilience of database systems = ability to recover from failures without loss.
- Logging as a protection against certain failures.
- Undo logging.

Soon:

- Checkpoints: avoiding logs of all history.
- Redo, and redo/undo logging.
- Transactions, serializability, locking, etc.: managing interactions among processes.

Architecture of the Optimizer

- Start with an algebraic representation of the query.
- Generate plans = order of operations + method details:

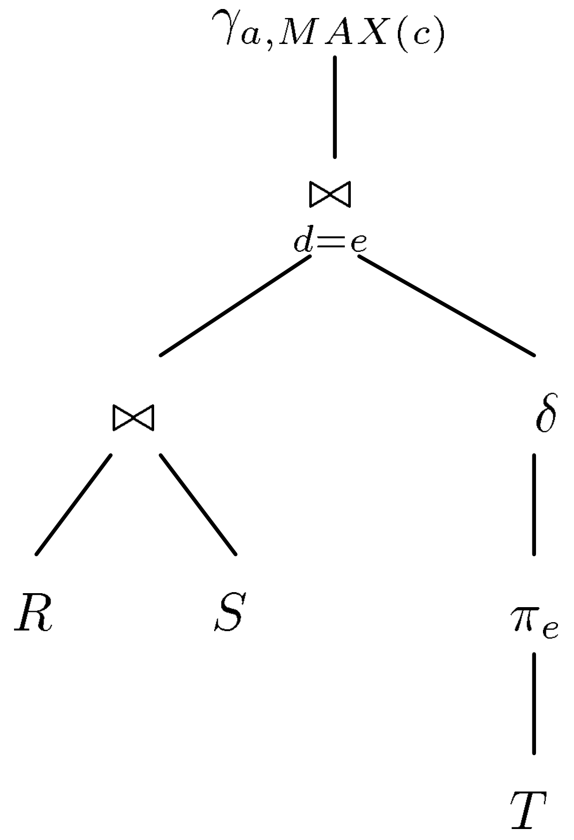
a) Heuristic.

b) Explore and estimate costs = *cost-based optimization.*

Example

Schema: $R(a, b)$, $S(b, c, d)$, $T(e, f)$.

```
SELECT a, MAX(c)
FROM R NATURAL JOIN S
WHERE EXISTS
    (SELECT * FROM T WHERE d=e)
GROUP BY a;
```



- Notice: `EXISTS` can be considered a join, but we *must* project and eliminate duplicates.

Pipelining Versus Materialization

Many operations (or certain implementations of them) allow us to *pipeline*, i.e., to accept one or both arguments in a stream without seeing the entire relation before starting.

- If not, then the argument must be *materialized* (stored on disk if it is large), before beginning.

Some examples:

- π, σ, δ, γ allow pipelining.
- Intersection does not allow pipelining.
- Nested-loop join allows the outer argument to be pipelined, but not the inner.
- Sort- or hash-join allows pipelining of either argument, but there are problems if both are:
  - Sort-join requires sharing of memory for sorting sublists.
  - Hash-join requires buffers for buckets of both relations in memory.
- Index join allows pipelining of the non-indexed argument.

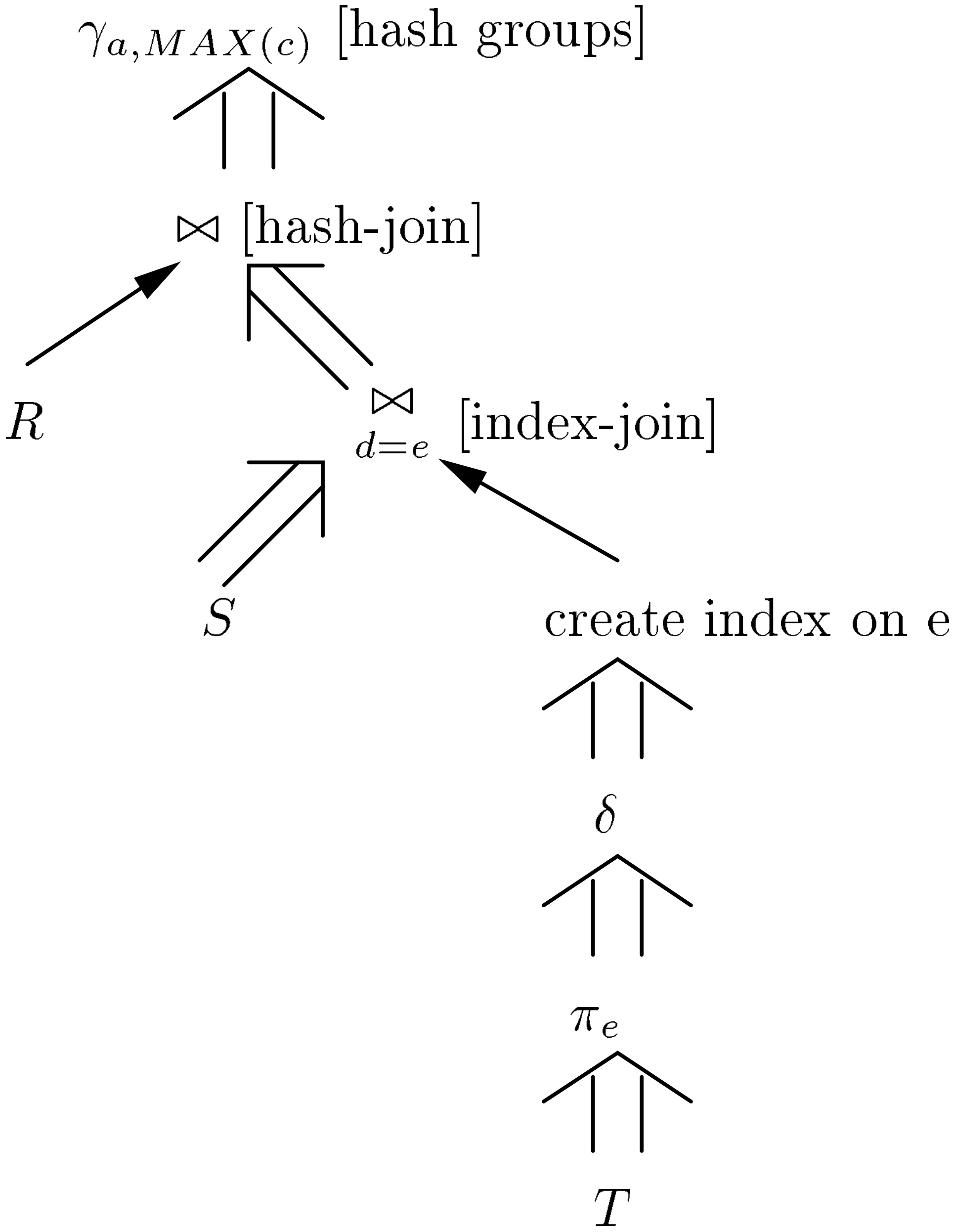
Notation

Double arrow = pipelined; single arrow = not.

One Possible Plan

Makes sense if T is small.

- But notice that the δ can be removed by a series of transformations if the cost analysis warrants it.


$\gamma_{a,MAX(c)}$ [hash groups]
$\bowtie$ [hash-join]
R
$\bowtie_{d=e}$ [index-join]
S
create index on e
δ
π_e
T

Cost-Based Optimization

- Enumerate all candidate expressions.
- Estimate sizes of intermediate expressions and the costs of constructing them.

Very Simple Example

$R(A, B, C)$; A is key, $N(R) = 10{,}000$; $V(B, R) = 1000$; $V(C, R) = 50$.

- 10 records/block; 100 main-memory buffers available.

```
SELECT DISTINCT B
FROM R
WHERE C = 1;
```

- Two feasible plans:

  1. $R \rightarrow \sigma_{C=1} \rightarrow \pi_B \rightarrow \delta$

  2. $R \rightarrow \pi_{B,C} \rightarrow \delta \rightarrow \sigma_{C=1} \rightarrow \pi_B$

- Plan (1) has estimated sizes 200 tuples (20 blocks) for each intermediate.

  - Thus, it can be pipelined, and the disk I/O's are only 1000 to read R and whatever is needed to write.

- Plan (2) will not substantially reduce the estimated size until the selection, so we must use 3000 disk I/O's for a 2PMMS.

  - It would be 4000, but we can pipeline the result of the sort to σ and π_B.

Cost of Joins

A very hard combinatorial problem appears when we must join a large number of relations. Approaches:

1. *Dynamic programming*: For each subset of the relations, compute the estimated size and estimated disk I/O cost of producing the result (a) pipelined (b) materialized.

   - In each case we can assume all buffers are available.

   - Dynamic-programming step consists of considering, for a subset S of size n, all possible ways to break S into two smaller sets, compute one materialized, and pipeline the other into one of several join methods.

2. *Left-deep trees*: Consider only those join expressions of the form $(\cdots((R_1 \bowtie R_2) \bowtie R_3) \bowtie \cdots \bowtie R_k)$.

   - Dynamic-programming step only needs to consider adding one relation to the chain.

Failure Modes

- *Erroneous data entry*: caught by DBMS through triggers, key, foreign-key, value- and tuple-based constraints, SQL2 assertions.
  - Requires modification of every query by the query-processor.
- *Media failures*:
  - Local: caught by parity checks on sectors, main memory.
  - Disk crash: handled by RAID.
- *System crash*: buggy program, power outage.
  - The issue for today.
  - Problem is that memory loss wipes out process state and data — what if the process had already made some modifications to the database?

Correctness Principle

A process that reads or modifies the DB is called a *transaction*. Basic assumptions:

- The database has a state (= values of all DB elements).
  - DB *element* is vague; it could be a tuple, block, whole relation, e.g.
- A state is "consistent" if it meets all constraints, both explicit (part of schema) and implicit.
- A transaction, if allowed to complete with no interference from other transactions or system errors, will take the DB from a consistent state to another consistent state.
- Conversely, a transaction is *atomic*. If only part of it executes, an error is likely to result.

Primitive DB Operations of Transactions

- `INPUT(X)` = copy the disk block containing database element X to a memory buffer.
- `READ(X,t)` = if the block containing database element X is not in a memory buffer then `INPUT(X)`. Next, assign the value of X to local variable t.
- `WRITE(X,t)` = if the block containing database element X is not in a memory buffer then `INPUT(X)`. Next, copy the value of t to X in the buffer.
- `OUTPUT(X)` = copy the buffer containing X to disk.

Example

- A, B are database values; constraint $A = B$ must hold.
- Transaction $T =$

```
A := A*2;
B := B*2;
```

- Execution of T involves reading A, B from disk, performing arithmetic in memory, and writing new A, B to disk.

`Action`	t	Mem A	Mem B	Disk A	Disk B
`READ(A,t)`	8	8		8	8
`t := t*2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t := t*2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`	16	16	16	16	8
`OUTPUT(B)`	16	16	16	16	16

- Problem: what happens if there is a system failure just before `OUTPUT(B)`?