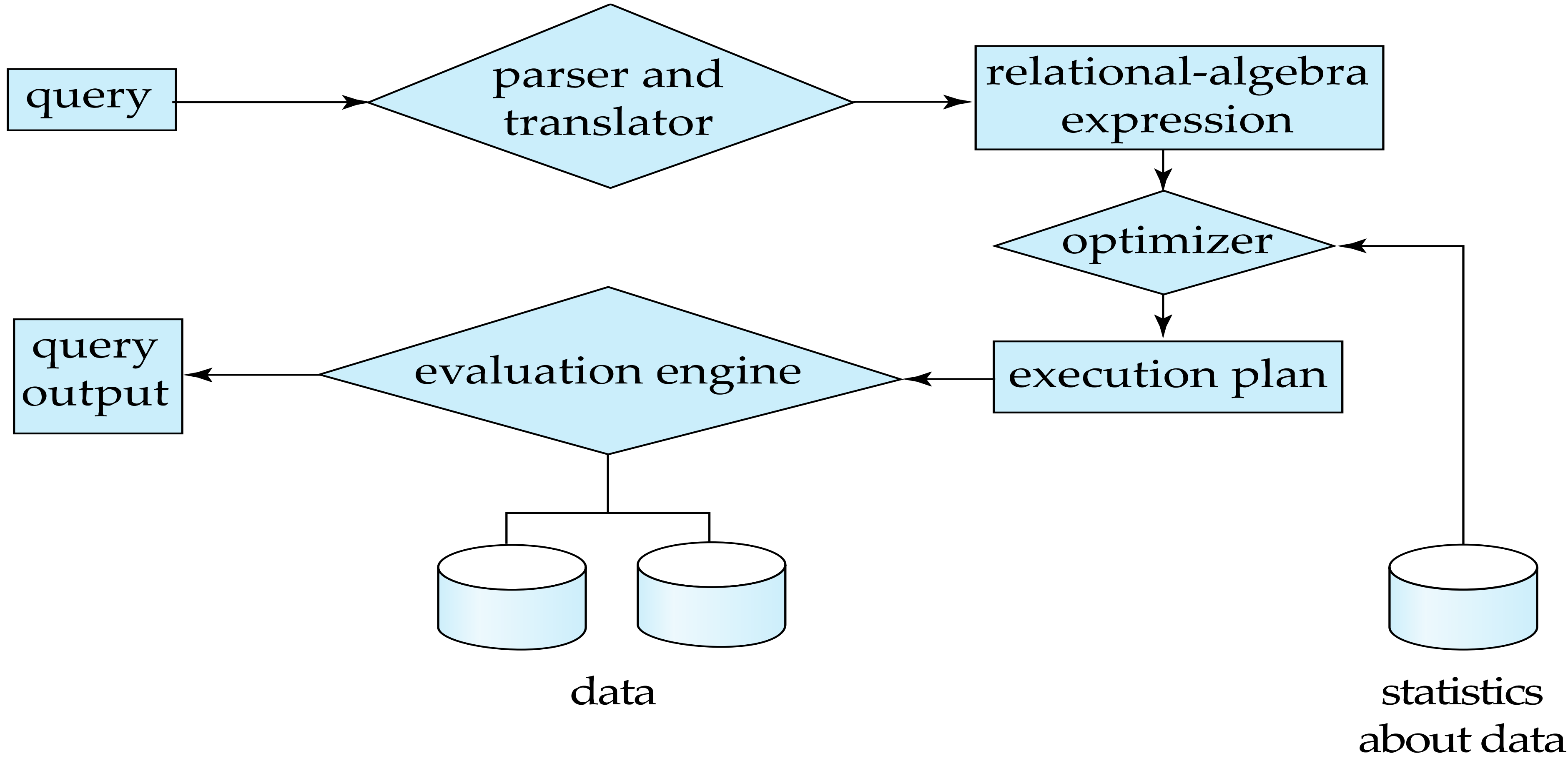
Query Processing

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

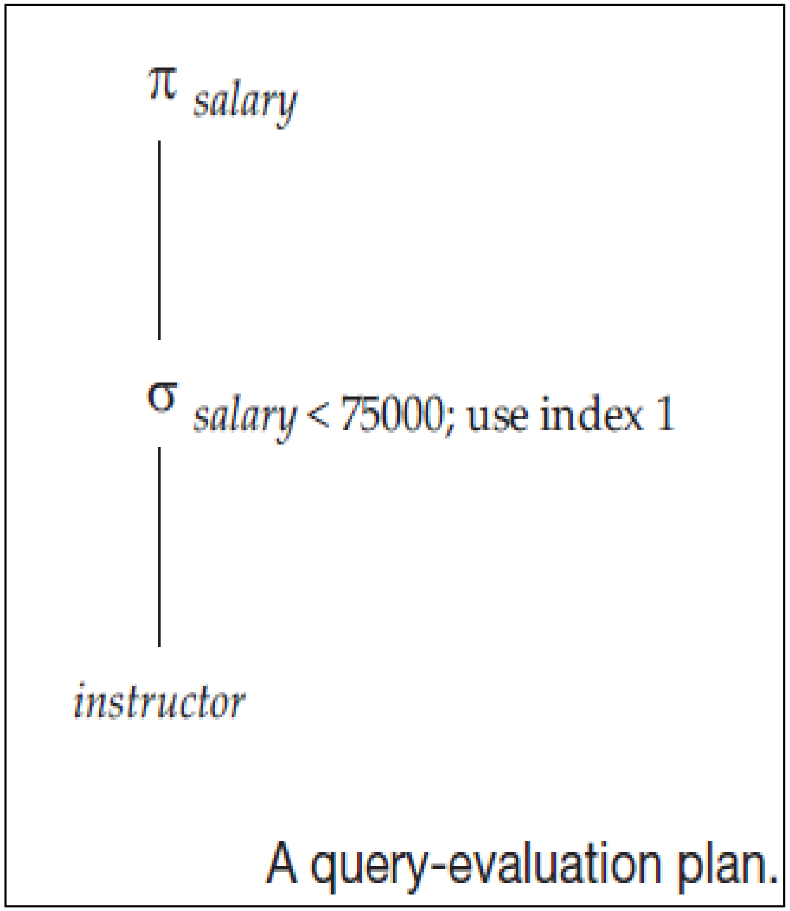
- Parsing and translation
  - Parser checks syntax, verifies relations.
  - The system constructs a parse-tree representation of the query,
  - translate the query into its internal form. This is then translated into relational algebra.
  - If the query was expressed in terms of a view, the translation phase also **replaces all uses of the view** by the relational-algebra expression.
- **Query Optimization**
  - The different evaluation plans for a given query can have different costs. It is the responsibility of the system to construct a query evaluation plan that **minimizes the cost of query evaluation**; this task is called **query optimization**.
- **Evaluation**
  - The query-execution engine takes a query-evaluation plan, **executes that plan**, and returns the answers to the query.

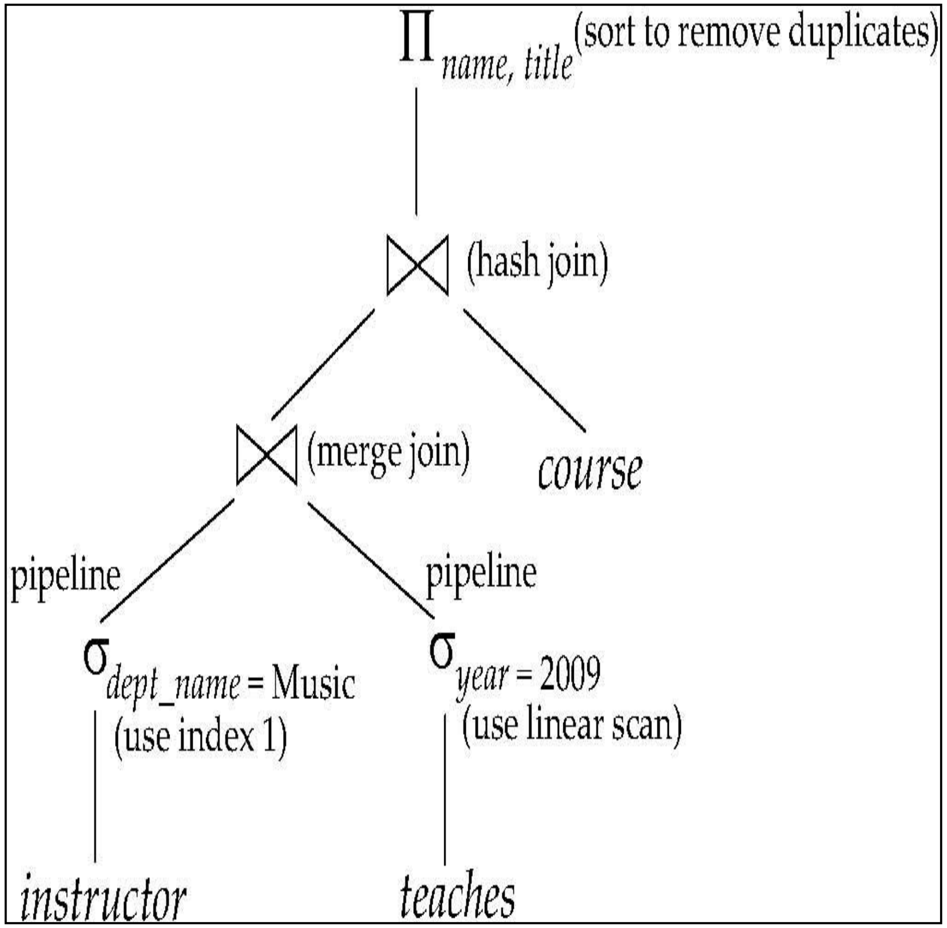
Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to $\prod_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use.
- An annotated instructions in a relational algebra is called an **evaluation primitive**.
- A **sequence of primitive operations** that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**.

Evaluation Plan

E.g., We can use an index on *salary* to find instructors with salary < 75000, **or** can perform complete relation scan and discard instructors with salary ≥ 75000




∏ name, title (sort to remove duplicates)
⋈ (hash join)
⋈ (merge join)
course
pipeline
pipeline
σ dept_name = Music (use index 1)
σ year = 2009 (use linear scan)
instructor
teaches

Annotated expression specifying detailed evaluation strategy is called an evaluation-plan

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - *data is read back after being written* to ensure that the write was successful

Measures of Query Cost (Cont.)

- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - We want to transfer **b** blocks of data and assume we require **S** seeks to transfer **b** blocks of data..
  - Cost for b block transfers plus S seeks

$$= b * t_T + S * t_S$$

- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account

Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.

  Cost estimate = b_r block transfers $*t_T + 1 * t_s$

    - b_r denotes number of blocks containing records from relation r

  - If selection is on a key attribute, can stop on finding record

    cost = $(b_r/2)$ block transfers $*t_T + 1 * t_s$

    - (Because -at most 1 record satisfies the search criteria)

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information ***student*** $\bowtie$ ***takes***
  - Number of **records** of *student*: n_r= 5,000

    takes: n_s= 10,000
  - Number of **blocks** of *student*: b_r= 100

    takes: b_s=400

Student

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

takes

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  For each record in r, we have to perform complete scan on s.
  pairs of tuples to be considered is $n_r * n_s$, where n_r denotes the number of tuples in r, and n_s denotes the number of tuples in s.

```
for each tuple $t_r$ in $r$ do begin
    for each tuple $t_s$ in $s$ do begin
        test pair $(t_r, t_s)$ to see if they satisfy the join condition $\theta$
        if they do, add $t_r \cdot t_s$ to the result.
    end
end
```

- *r* is called the **outer relation** and ***s*** the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

  $n_r * b_s + b_r$ block transfers, and

  $n_r + b_r$ seeks*

records of	*student*:	n_r = 5,000
	takes:	n_s = 10,000
blocks of	*student*:	b_r = 100
	takes:	b_s =400

- Example: Assuming **worst case** memory availability cost estimate is
  - with **student as outer relation**: (student has 100 blocks, smaller relation than takes, hence costlier)
    - 5000 ∗ 400 + 100 **= 2,000,100** block transfers,
    - 5000 + 100 **= 5100** seeks,

Nested-Loop Join (Cont.)

records of	*student*:	n_r= 5,000
	takes:	n_s= 10,000
blocks of	*student*:	b_r= 100
	takes:	b_s=400

- with ***takes*** **as the outer relation** (takes has 400 blocks, bigger relation than student, The number of block transfers is significantly less, and although the number of seeks is higher, the overall cost is reduced, assuming seek time **t_S = 4 milliseconds** and transfer time **t_T = 0.1 milliseconds.** hence costlier)
  - 10000 ∗ 100 + 400 = **1,000,400** block transfers and **10,400** seeks

Nested-Loop Join (Cont.)

- If the **any one relation fits entirely in memory**, use that as the **inner relation.**
  - Reduces cost to **b_r + b_s block transfers** and **2 seeks**
- If **smaller relation (*student*) fits entirely in memory**, the cost estimate will be 500 block transfers.

*** See notes section.**

Block Nested-Loop Join

- **Variant** of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block $B_r$ of $r$ do begin
    for each block $B_s$ of $s$ do begin
        for each tuple $t_r$ in $B_r$ do begin
            for each tuple $t_s$ in $B_s$ do begin
                Check if $(t_r, t_s)$ satisfy the join condition
                    if they do, add $t_r \bullet t_s$ to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- **Worst case** estimate: $b_r * b_s + b_r$ block transfers and **2** $* b_r$ seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation.

- **Best case:** If the smaller relation fits entirely in memory
  - $b_r + b_s$ block transfers & **2** seeks.

Example of Block Nested-Loop Join Costs

- Compute ***student*** ⋈ ***takes***, with *student* as the outer relation.
  - Number of records of *student*: $n_{student}$ =**n$_r$= 5, 000**.
  - Number of blocks of *student*: $b_{student}$ = **b$_r$=100.**
  - Number of records of *takes*: n_{takes} =**n$_s$= 10, 000**.
  - Number of blocks of *takes*: b_{takes} = **b$_s$ =400.**
- Cost of block nested loops join

  cost $= b_r * b_s + b_r$ block transfers and $2 * b_r$ seeks
  - 400*100 + 100 = 40,100 block transfers and
  - 2 * 100 = 200 seeks
    - assuming worst case memory
    - may be significantly less with more memory

Evaluation of Expressions

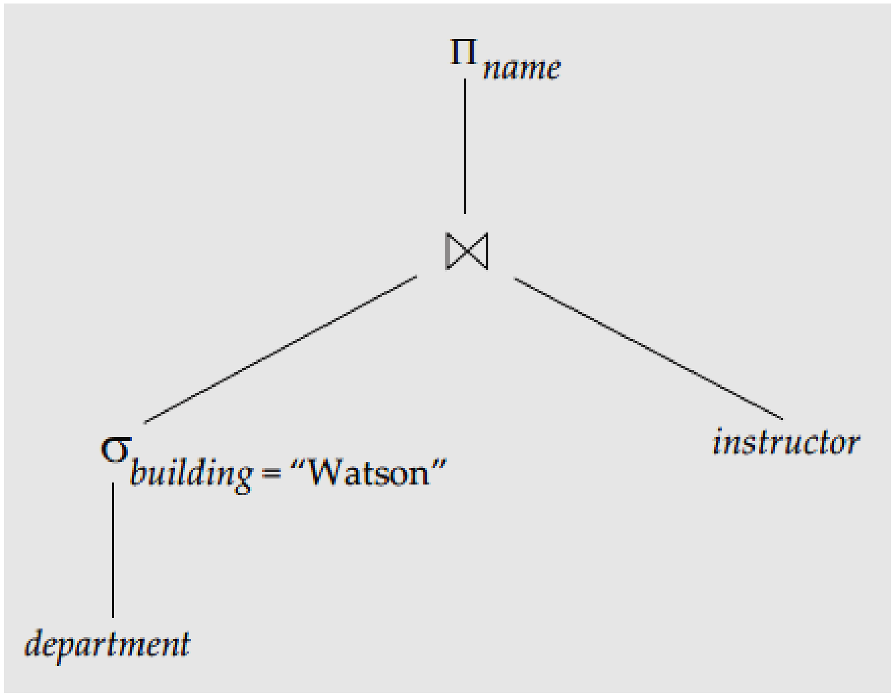
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="Watson"}(department)$$

- then compute the store its join with *instructor,* and finally compute the projection on *name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high.
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- We assume that the records of the result accumulate in a buffer, and, when the buffer is full, they are written to disk
- The number of blocks written out, b_r, can be estimated as n_r / f_r, where n_r is the estimated number of tuples in the result relation r, and f_r is the blocking factor of the result relation, that is, the number of records of r that will fit in a block

Materialization (Cont.)

- In addition to the transfer time, some disk seeks may be required, since the disk head may have moved between successive writes.
- The number of seeks can be estimated as $\lceil b_r / b_b \rceil$ where b_b is the size of the output buffer (measured in blocks).
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled by executing algorithm.
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation :** evaluate several **operations simultaneously**, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\boxed{\sigma_{building="Watson"}(department)}$$

  - instead, **pass tuples directly to the join**
  - Similarly, don't store result of join, **pass tuples directly to projection**.
- **Two benefits**
  - It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation
  - It can start generating **query results quickly**, if the root operator of a query evaluation plan is combined in a pipeline with its inputs.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation (Pull)
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining (Push)
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

-Pipelining (Cont.)

- Implementation of demand-driven pipelining
- Each operation(Selection, Join) is implemented as an **iterator** and **iterator** has the following operations(functions)
  - **open()**
    - E.g. file scan: initialize file scan
      - state: pointer to beginning of file
    - E.g.merge join: sort relations;
      - state: pointers to beginning of sorted relations
  - **next()**
    - E.g. for file scan: Output next tuple, and advance and store file pointer
    - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
  - **close()**

-Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - intermediate results written to disk and then read back
- Algorithm variants to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in

Materialized Views

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view

  create view *department_total_salary*(*dept_name, total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*

- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can **be maintained by re-computation on every update**
- A better option is to use **incremental view maintenance**
  - modify only the affected parts of the materialized view,
  - which is known as **incremental view maintenance**
- **View** maintenance can be done by
  - **Immediate view maintenance**
    - Manually **defining triggers on insert, delete, and update** of each relation in the view definition
    - Manually written code to update the view whenever database relations are updated
  - **Deferred view maintenance**
    - **Periodic re-computation** (**e.g. nightly**)
    - Above methods are directly supported by many database systems

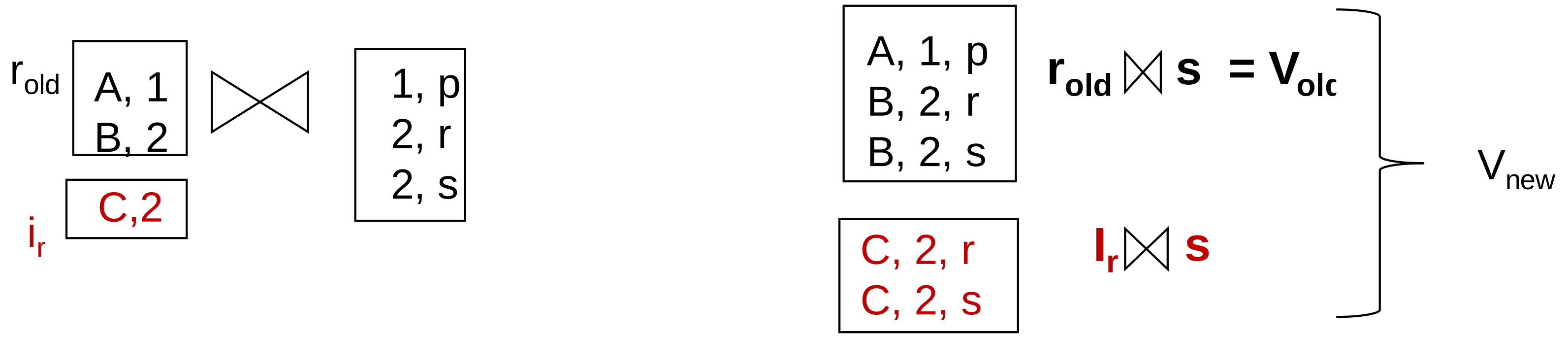
Incremental View Maintenance

- The **changes (inserts and deletes**) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from **r** are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- How to do maintenance of materialized view created by JOIN.
- Consider the **materialized view $v = r \bowtie s$** and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert i_r *tuples* to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

s



r_{old}
A, 1
B, 2
i_r
C,2
1, p
2, r
2, s
A, 1, p
B, 2, r
B, 2, s
$r_{old} \bowtie s = V_{old}$
C, 2, r
C, 2, s
$I_r \bowtie s$
V_{new}