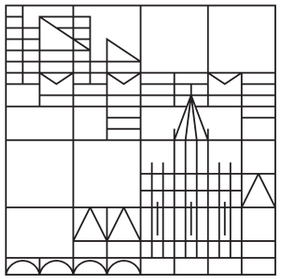
Universität
Konstanz



Database System Architecture and Implementation

Module 5

Query Evaluation Overview

November 20, 2018

Orientation

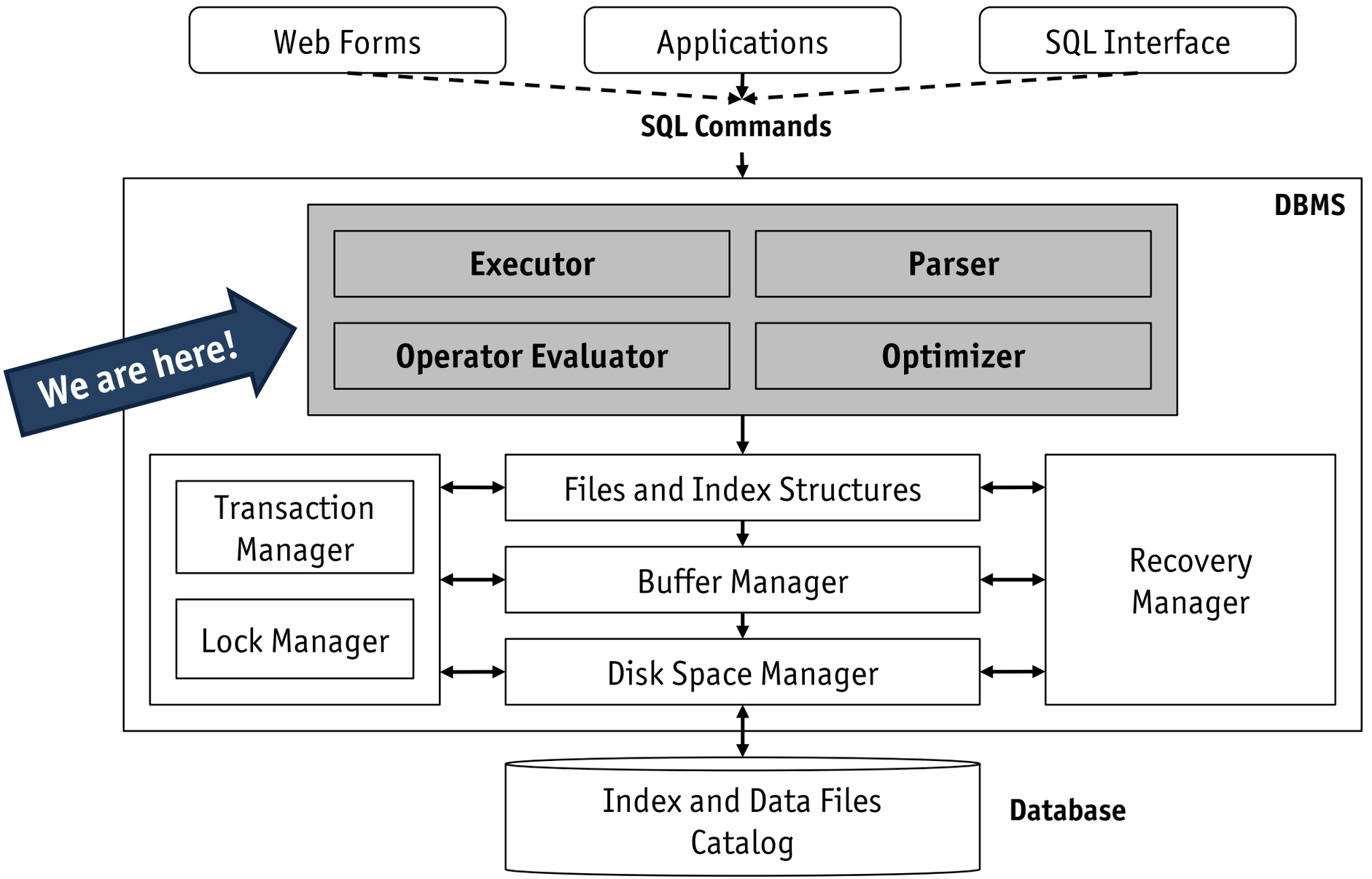


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- System catalog
- Relational operator evaluation
  - algorithms for σ, π, and $\bowtie$
  - operator scheduling
- Query optimization
  - query evaluation plans
  - cost estimation of a plan

Running Example

A simple schema

```
CREATE TABLE Sailors (
  sid      INTEGER,
  sname    STRING,
  rating   INTEGER,
  age      REAL,
  PRIMARY KEY (sid)
)
```

```
CREATE Table Reserves (
  sid      INTEGER,
  bid      INTEGER,
  day      DATE,
  rname    STRING,
  PRIMARY KEY (sid, bid, day),
  FOREIGN KEY sid REFERENCES
    Sailors(sid)
)
```

- Assumptions
  - relation **`Sailors`**: 50 bytes/tuple, 80 tuples/page, and 500 pages
  - relation **`Reserves`**: 40 bytes/tuple, 100 tuples/page, and 1000 pages

↳ 4 kB page size, 2 MB **`Sailors`** data, and 4 MB **`Reserves`** data

System Catalog

- A DBMS manages **two types of information**
- Data
  - DBMS uses several alternative file structures to store **tables** and **indexes**
  - conversely, files contains either **tuples of table** or **index entries**
  - collection of user tables and indexes represents that **data** of the database
- Metadata
  - DBMS also maintains information that **describes** every table and index
  - descriptive information itself stored in a collection of **special tables**
  - this **metadata** is called catalog tables, data dictionary, or system catalog

System Catalog

- System catalog stores system-wide information
  - **size of buffer pool**, the **page size**, etc.
  - information about **tables**, **views**, and **indexes**.

Information stored in the system catalog

- **Table metadata**
  - *table name*, *file name* (or some identifier), *file structure* (e.g., heap file)
  - *attribute name* and *type* of each attribute of the table
  - *index name* of each index on the table
  - *integrity constrains* (e.g., primary and foreign key constraints) on the table
- **Index metadata**
  - *index name* and *structure* (e.g., B+ tree)
  - *search key* attributes
- **View metadata**
  - *view name* and *definition*

System Catalog

- Additionally, the system catalog stores
  - **statistics** about tables and indexes
  - statistics are updated **periodically**, **not every time** tables are modified

Statistics

- **Table statistics**
  - *cardinality*: number of tuples *NTuples*(*R*) for each table *R*
  - *size*: number of pages *NPages*(*R*) for each table *R*
- **Index statistics**
  - *cardinality*: number of distinct key values *NKeys*(*I*) for each index *I*
  - *size*: number of pages *INPages*(*I*) for each index (for a tree index *I*, *INPages*(*I*) denotes the number of leaf pages)
  - *height*: number of non-leaf levels for each tree index *I*
  - *range*: minimum present key value *ILow*(*I*) and the maximum present key value *IHigh*(*I*) for each index *I*

System Catalog

What else?

Can you think of other information that would be useful to keep in the system catalog?

System Catalog

- DBMS stores system catalog **itself** as a collection of tables
  - **existing techniques** for implementing and managing tables are reused
  - **same query language** can be used for catalog tables as for other tables

Metamodeling

The technique to model a data model in itself is known as **metamodeling** and the resulting data model is referred to as a **metamodel.** Since the choice of catalog tables and their schemas is not unique, real DBMS vary in terms of the actual metamodel.

As the designer of a DBMS, how would you design the schema of the system catalog?

System Catalog

Example

Tables

name	file	#tuples	size
Tables	...	6	1
Attributes	...	23	1
Views	...	1	1
Indexes	...	0	1
Sailors	...	40,000	500
Reserves	...	100,000	1000

Views

name	text
Captains	SELECT * FROM Sailors WHERE...

Indexes

name	file	type	#keys	size
Boats	...	B+Tree	100	1

Attributes

name	table	type	pos
name	Tables	string	1
file	Tables	string	2
#tuples	Tables	integer	3
size	Tables	integer	4
name	Attributes	string	1
table	Attributes	string	2
type	Attributes	string	3
pos	Attributes	integer	4
⋮	⋮	⋮	⋮
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	date	3
rname	Reserves	string	4

System Catalog

The Real World

- **Oracle 10g** defines a total of 8 system tables, including...
  - **`ALL_TABLES`**: *owner*, *table name*, and about 50 other attributes
  - **`ALL_VIEWS`**: *owner*, *view name*, *view text* and about 10 other attributes
  - **`ALL_TAB_COLUMNS`**: *owner*, *table name*, *column name*, *data type*, *length*, *nullable*, *default*, *low*, *high*, *density*, *histogram* and about 25 other attributes
- **Microsoft SQL Server**
  - **`sys.objects`**: all user-defined, schema-scoped objects in a database in terms of *name*, *id*, *type* (e.g., **`U`** = user-defined table) and about 10 other attributes
  - **`OBJECT_ID(`***name*, *type*`)`**:** meta data function that returns *id* of an object
- **ANSI SQL-92**
  - **`INFORMATION_SCHEMA`**: provides information about *tables*, *views*, *columns* and *procedures*
  - many system implement the standard by defining **views** over their proprietary system catalog tables

System Catalog

Microsoft SQL Server: schema definition in Transact-SQL

```
IF OBJECT_ID ('dbo.Employees', 'U') IS NOT NULL
  DROP TABLE dbo.Employees
IF OBJECT_ID ('dbo.Managers', 'V') IS NOT NULL
  DROP VIEW dbo.Managers
IF OBJECT_ID ('dbo.udf_DistributeBonus', 'P') IS NOT NULL
  DROP PROCEDURE dbo.udf_DistributeBonus
IF OBJECT_ID ('dbo.udf_CalculateBonus', 'FN') IS NOT NULL
  DROP FUNCTION dbo.udf_CalculateBonus
GO

CREATE TABLE dbo.Employees ( ... )
CREATE VIEW dbo.Managers AS ...
CREATE PROCEDURE dbo.DistributeBonus AS ...
CREATE FUNCTION dbo.udf_CalculateBonus(@empId INT)
  RETURNS FLOAT BEGIN ... END
```

Relational Operator Evaluation

- Several **alternative algorithms** can be used to implement each relational operators
  - for most operators no algorithm has universally superior performance
  - performance depends on sizes of involved tables, existing indexes and sort orders, size of buffer pool and buffer replacement policy
- Alternative algorithms for relational operators often use **common implementation techniques**
  - **indexing:** an index is used to only process tuples that satisfy a selection or join condition
  - **iteration:** process all tuples of an input table, one after the other, or scan all index data entries of an index that contains all required attributes (not using the search structure of the index)
  - **partitioning:** decompose an operation into a less expensive collection of operations by partitioning tuples on a sort key (e.g., sorting and hashing)

Access Paths

- An **access path** is a way of retrieving tuples from a table, e.g.,
  - file scan
  - index plus a matching selection condition
- Every relational operator accepts **one or more tables** as input
- Access paths chosen to retrieve tuples contribute significantly to the **cost of the operator**

Selection conditions

A selection is in **conjunctive normal form (CNF)**, if it is a conjunction ($\wedge$) of conditions of the form

attribute **op** *value*,

where **op** is one of the comparison operators $<$, $\leq$, $=$, $\neq$, $\geq$, or $>$, and each of these conditions is called a **conjunct**.

Access Paths

- An index **matches** a selection condition, if it can be used to retrieve just the tuples that satisfy the condition
  - a **hash index** matches a CNF selection, if there is a conjunct of the form *attribute* = *value* for each attribute in the index's search key
  - a **tree index** matches a CNF selection, if there is a conjunct of the form *attribute* **op** *value* for each attribute in a prefix of the index's search key

Example: search key prefixes

$\langle a \rangle$, $\langle a, b \rangle$ are prefixes of key $\langle a, b, c \rangle$, whereas $\langle a, c \rangle$ and $\langle b, c \rangle$ are not.

- An index can match some **subset** of the conjuncts of a CNF selection, even though it does not match the entire selection
  - **primary conjuncts:** the conjuncts that the index matches

Access Paths

Exercise: hash index vs. B+ tree index

Assume a **hash index** *H* on the search key ⟨*rname, bid, sid*⟩ on table **`Reserves`**. Which of the following CNF conditions does *H* match?

- *rname* = 'Joe' ∧ *bid* = 5 ∧ *sid* = 3
- *rname* = 'Joe' ∧ *bid* = 5
- some condition on *day*

Assume a **B+ tree index** *I* on the search key ⟨*rname, bid, sid*⟩ on table **`Reserves`**. Which of the following CNF conditions does *I* match?

- *rname* = 'Joe' ∧ *bid* = 5 ∧ *sid* = 3
- *rname* = 'Joe' ∧ *bid* = 5
- *bid* = 5 ∧ *sid* = 3

Access Paths

Example: primary conjuncts

Assume an index (hash or B+ tree) on the search key ⟨*bid, sid*⟩ on table **`Reserves`** and the selection condition *rname* = 'Joe' ∧ *bid* = 5 ∧ *sid* = 3

1. the index can be **scanned** to retrieve the tuples that satisfy the condition *bid* = 5 ∧ *sid* = 3, i.e., the primary conjuncts
2. the **additional condition** on *rname* must then be applied to each retrieved tuple and will eliminate some of the retrieved tuples from the result.

↳ the **number of pages** retrieved depends on the fraction of tuples that satisfy these conjuncts and whether the index is clustered or not

Access Paths

Example: two (partially) matching indexes

Assume an index (hash or B+ tree) on the search key ⟨*bid, sid*⟩, a B+ tree index on *day*, and the selection condition $day < 8/1/2013 \wedge bid = 5 \wedge sid = 3$

- both indexes match (part of) the selection condition and either one can be used to retrieve **`Reserves`** tuples
- regardless of whichever is used, the conjuncts in the selection condition that are not matched by the index must be checked for each retrieved tuple

↳ if the B+ tree index on *day* is used, the condition $bid = 5 \wedge sid = 3$ must be checked
↳ if the hash index on ⟨*bid, sid*⟩ is used, the condition $day < 8/1/2013$ must be checked

Access Path Selectivity

- Recall that several alternative access paths may exist to retrieve all desired tuples in a table
  - **scan** of the data file
  - **index**, if table has an index that matches selection given in the query
  - **index scan**, if index contains all attributes required by the query

Definition: selectivity of an access path

- the **selectivity** of an access path is the number of index and data pages it retrieves
- the **most selective** access path is the one that retrieves the fewest pages

↳ choosing the most selective path minimizes cost of data retrieval, i.e., number of I/O operations

Access Path Selectivity

- Apart from data file organizations, the selection condition (with respect to the index involved) affects access path selectivity
  - selectivity depends on the **primary conjuncts** in the selection condition
  - each conjunct acts as a **filter** on the table

Definition: reduction factor

- the fraction of tuples in the table that satisfy a given conjunct is called the **reduction factor**.
- if there are several primary conjuncts, the fraction that satisfies all of them can be **approximated** by the product of their reduction factors.

↳ this approach effectively treats all conjuncts as **independent** filters
↳ while they **may not be independent**, this approximation is widely used in practice

Access Path Selectivity

Example

Assume a **hash index** *H* on table **`Reserves`** with search key ⟨*rname, bid, sid*⟩ and the selection condition *rname* = 'Joe' ∧ *bid* = 5 ∧ *sid* = 3

- *H* can be used to retrieve tuples that satisfy **all** conjuncts
- the catalog contains the **number of distinct keys** *NKeys*(*H*) in the hash index
- the catalog also contains total number of pages *NPages* in the **`Reserves`** table

↳ the **reduction factor**, i.e., the fraction of tuples satisfying the primary conjuncts is

$$NPages(Reserves) \cdot \frac{1}{NKeys(H)}$$

Access Path Selectivity

Exercise: partially matching indexes

How would you estimate the fraction of tuples that satisfy a selection condition, if no index matches the entire condition?

Access Path Selectivity

- Reduction factor estimation for range selection conditions
  - assume that range attribute values are **uniformly** distributed
  - catalog information *ILow* and *IHigh* can be used
- If there is a tree index *T*, the estimated reduction factor is
  - $\frac{IHigh(T)-value}{IHigh(T)-ILow(T)}$ for selection condition *attr* < *value*.

Relational Operator Evaluation

- Focus on σ, π, and ⋈ operator as used in so-called **SPJ queries**
  - **this week:** brief overview of evaluation algorithms for these operators
  - **next two weeks:** in-depth look at sorting and operator evaluation
- Cost analysis
  - as before, **only I/O costs** are considered
  - I/O costs are measured in terms of **number of page I/O operations**
  - **examples**, rather than rigorous cost formulas

Selection

- Selection given as $\sigma_{R.attr\ \mathbf{op}\ value}(R)$
  - if there is **no index** on *R.attr*, R has to be scanned
  - if **one or more indexes** match the selection, an index can be used (possibly followed by application of remaining selection conditions)

Example: selection *rname* < 'C%' on `Reserves` table

- assuming that names are **uniformly distributed** with respect to the initial letter, roughly 10% of **`Reserves`** tuples are estimated to be in the result (for simplicity)
- since **`Reserves`** has 100,000 tuples, this is a total of 10,000 tuples or 100 pages
- if there is a **clustered** B+ tree index on **`Reserves`**.*rname*, the qualifying tuples can be retrieved with 100 I/O operations (plus a few I/O operations to traverse the tree)
- if the index is **unclustered**, retrieving the tuples could require 10,000 I/O operations

A rule of thumb

If over 5% of the tuples of a table are to be retrieved, it is **likely to be cheaper** to simply scan the entire table (instead of using an unclustered index)

Projection

- Projection given as $\pi_{\{R.attr1,\ R.attr2,\ \ldots\}}(R)$
  - dropping attributes from the input is the easy part
  - the expensive part is **duplicate elimination** from the result
- **Without** duplicate elimination (no `DISTINCT` in `SELECT`)
  - **scan of the table** to retrieve the projected subset of attributes
  - **scan of an index** whose key contains all necessary attributes (clustered vs. unclustered does not matter as only key values are retrieved)
- **With** duplicate elimination (by partitioning)
  1. as above, retrieve subset of projected attributes using a **scan**
  2. using the projected attributes as sort key, **sort** all retrieved tuples
  3. scan tuples and **discard** adjacent duplicates

Projection

- Sorting **disk-resident** data sets is next week's topic
  - typically requires two or three passes of reading and writing entire table
  - projection can be **optimized** by combining it with sorting

Optimized projection (with duplicate elimination)

- **first pass of sorting** scans entire table, but only writes out the subsets of projected attributes
- there might be an **intermediate pass** in which all subsets of projected attributes are read from and written to disk
- **final pass of sorting** scans all tuples, but only one copy of each subset of projected attributes is written out

Projection

- Availability of appropriate indexes can lead to **less expensive** plans than sorting for duplicate elimination
  - if an index exists whose **search key contains all attributes** retained by the projection, we can sort the index entries, rather than the data records
  - if all retained attributes appear in a **prefix of the search key** of a tree index, duplicates are easily detected since they are adjacent

↳ These are examples of **index-only** evaluation strategies

Join

- Join given as $R \bowtie_{R.attr = S.attr} S$
  - as joins are both **common** and **expensive**, they have been widely studied
  - DBMS typically supports **several algorithms** to compute joins

Index nested loops join

Consider a join of R and S with the condition $R.attr = S.attr$. If one of the tables has an index on column *attr,* an **index nested loops join** can be used.

Index nested loops join (sketch)

```
function join(R, I, attr)
  for each tuple t_R in R do
    do
      t_S = probe I with key t_R.sid;
      if t_S != null then emit t_R ⋈ t_S;
    until t_S == null;
end;
```

Join

- Assume that *I* is a hash-based index and it takes about **1.2 I/O operations** on average to retrieve the appropriate page
  - suppose index uses variant ❷, i.e., contains ⟨*k*, *rid*⟩ entries
  - since *sid* is a key for **`Sailors`**, there is **at most** one matching tuple
  - since *sid* is a foreign key in **`Reserves`**, there is exactly one match

Cost analysis of index nested loops join plan

- cost of scanning **`Reserves`** is 1000 (pages)
- there are 100 · 1000 tuples in **`Reserves`**
- for each of these tuples, retrieving the index page containing the corresponding *rid* of the matching **`Sailors`** tuple costs 1.2 I/O operations on average
- additionally, the **`Sailors`** page containing the qualifying tuple must be retrieved

↳ **total cost** is 1000 + (1.2 + 1) · 100,000 = 221,000 I/O operations

Join

- If there is no index that matches the join condition on either table, the index nested loop join cannot be used

Sort-merge join

If no index matches the join condition, a **sort-merge join** can be used

1. **sort** both tables on the join column
2. **scan** sorted tables to find matches

Index nested loop join (very high-level sketch)

```
function join(R, S, attr)
   sort R on attr; ↑t_R = first tuple in R;
    sort S on attr; ↑t_S = first tuple in S;
   do
      if t_R.attr != t_S.attr then advance t_R else emit t_R ⋈ t_S and advance t_S;
   until t_R and t_S reach the end of the table R and S, respectively;
end;
```

Join

- Assume that both tables can each be sorted in **two** (!) passes

Cost analysis of sort-merge join plan

- sort of **`Reserves`** reads and writes all of its pages two times: $2 \cdot 2 \cdot 1000 = 4000$
- sort of **`Sailors`** reads and writes all of its pages two times: $2 \cdot 2 \cdot 500 = 2000$
- second phase of sort merge join requires an additional scan of both tables, i.e., $1000 + 500 = 1500$

↳ **total cost** is $4000 + 2000 + 1500 = \underline{7500}$ I/O operations

- Remarks
  - cost of sort-merge join (which **does not require** a pre-existing index) is lower than cost of index nested loops join
  - additionally, the result of the sort-merge join is sorted on join column
  - other join algorithms that do not rely on existing indexes and are often cheaper than index nested loops joins are also known

Join

Exercise: why consider index nested loops joins at all?

What nice property does an index nested loops join have that a sort-merge join does not have with respect to cost?

Other Operators

- In addition to the basic relational operators (σ, π, and $\bowtie$), SQL queries can also contain
  - **group by**
  - **aggregation**
  - **set operations**: union, difference, and intersection
- Set operations
  - as in projection, expensive aspect is **duplicate elimination**
  - **partitioning approach** for projection can be adapted for these operations
- Group by
  - typically implemented through **sorting**
  - sorting step can be avoided, if a **tree index** with the grouping attributes as search key exists
- Aggregate operators are carried out using **temporary counters** in main memory as tuples are retrieved

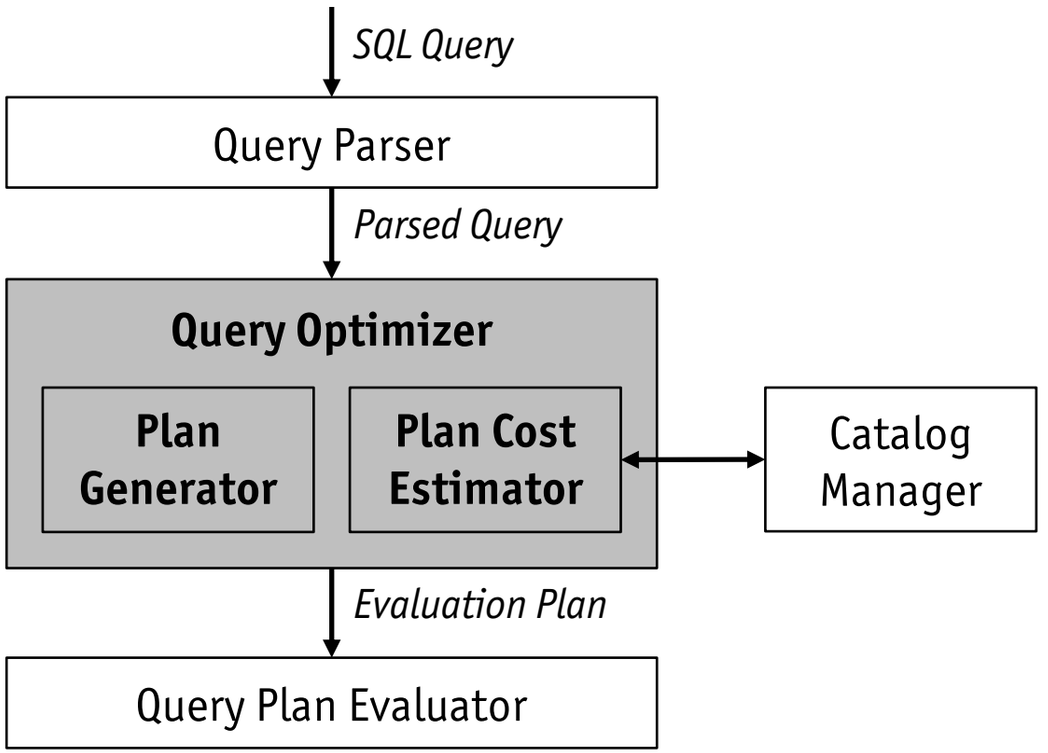
Query Optimization

- *"With great power comes great responsibility."* (Ben Parker)
  - SQL gives users **great flexibility** how to (declaratively) express a query
  - DBMS can chose from **many different strategies** to evaluate a query
- Quality of the **query optimizer** greatly influences performance
  - difference in cost between the best and worst evaluation strategy may be **several orders of magnitudes**
  - query optimizer cannot be expected to always find the best strategy, but it should consistently find a strategy that is **quite good**

Commercial optimizers

Current relational DBMS optimizers are **very complex** pieces of software with many **closely guarded details**, and they typically represent **40 to 50 man-years** of development effort.

Query Optimization



- Query optimizer identifies an efficient execution plan
  - **plan generator** enumerates alternative plans
  - **plan cost estimator** assigns a cost of each alternative plan
  - plan with the least estimated cost is chosen

Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Query Optimization

- To enumerate alternative plans, a query optimizer explores the **search space** of possible plans
  - queries are essentially treated as **σ-π-⋈ algebra expressions**
  - **remaining operations** are applied to the result of the σ-π-⋈ expression
- Optimizing a σ-π-⋈ algebra expression involves three steps
  1. **enumerating** alternative plans for evaluating the expression
  2. **estimating** the cost of each enumerated plan
  3. **choosing** the plan with the lowest estimated cost

Restricting the search space

Typically, a query optimizer considers a **subset of all possible plans** because the number of possible plans is very large.

Query Evaluation Plans

- A **query evaluation plan** (or simply plan) consists of an relational algebra tree extended with annotations at each node
  - access paths to use for each table
  - algorithm to use for each relational operator

Example: a simple SPJ query

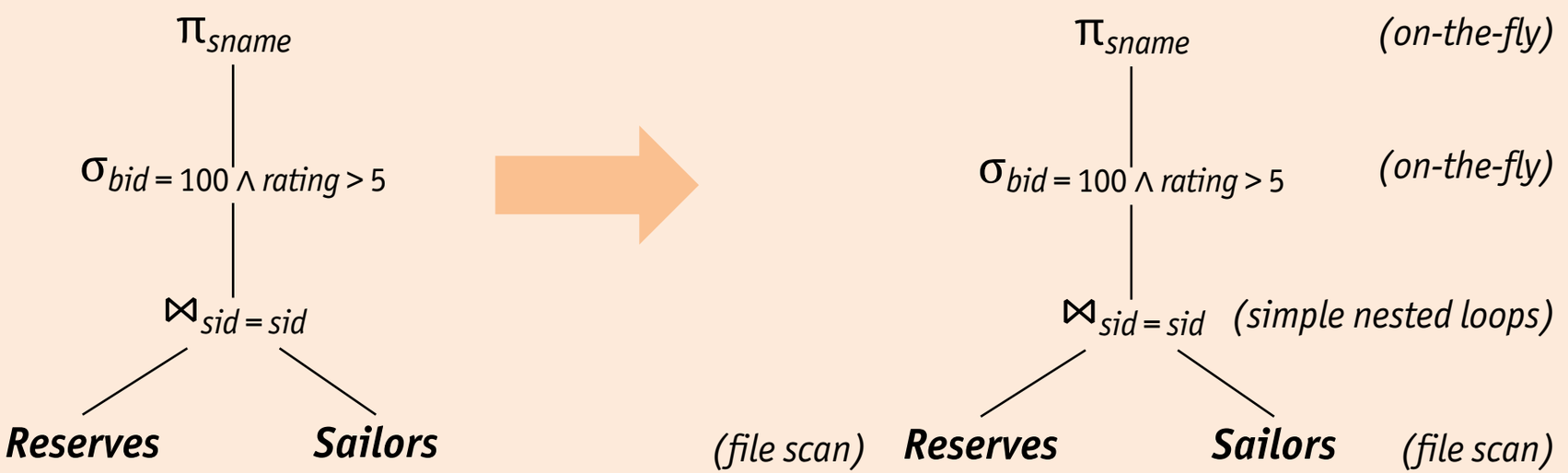
```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND R.bid = 100 AND S.rating > 5
```

Exercise: translate from SQL to relational algebra

Express the SQL given above in relational algebra using σ, π, and ⋈ operators.

Query Evaluation Plans

Example: from relational algebra tree to query evaluation plan



- Remarks
  - join operator uses the page-oriented **simple nested loops join** algorithm
  - by convention the **outer table is the left child** of the join operator
  - selections and projections are applied **on-the-fly** to each tuple in the result of the join as it is produced

Multi-Operator Queries

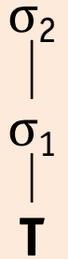
- If a query is composed of **several operators**, the query evaluator needs to decide how these operators
  - **communicate** with each other to pass results of one operator to the next
  - are **scheduled** and **interleaved** (i.e., inter-operator parallelism)
- Communication
  - **materialized:** operator writes its output to a temporary table, next operator reads its input from this temporary table
  - **pipelined:** results are directly "streamed" (typically, one page at a time) from one operator the next as they are produced
- Scheduling
  - **bracket model:** explicit scheduling
  - **iterator model:** implicit scheduling

Materialized vs. Pipelined Evaluation

- Pipelining has **lower overhead costs** than materialization
  - avoids cost of writing out intermediate results and reading them back in
  - chosen whenever algorithm for operator evaluation permits it
- There are **many opportunities** for pipelining in typical query plans, even simple plans that only involve selections

Example: selection-only query

Consider a selection-only query in which only part of the selection condition matches an index: we can think of such a query as containing **two** instances of the selection operator (σ)



- the first selection (σ_1) contains the **primary** (or matching) **part** of the original selection condition
- the second selection (σ_2) contains the **rest** of the selection condition

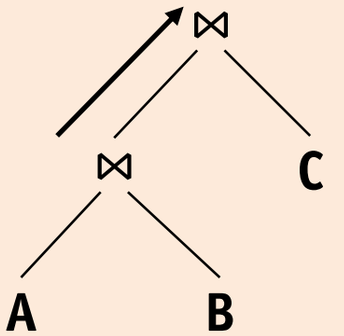
↳ **materialized:** apply σ_1 and write results to a temporary table, apply σ_2 to this table

↳ **pipelined:** apply σ_2 to each tuple in the result of σ_1 as it is produced

Materialized vs. Pipelined Evaluation

Example: joins

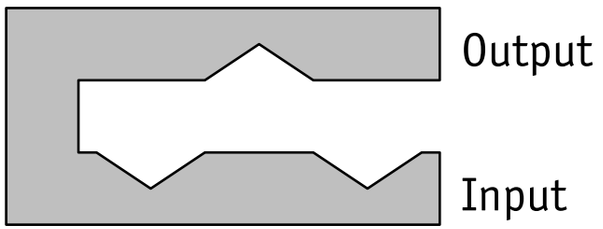
Consider a join of the form (A ⋈ B) ⋈ C: both joins can be evaluated in pipelined fashion using some form of **nested loops join**



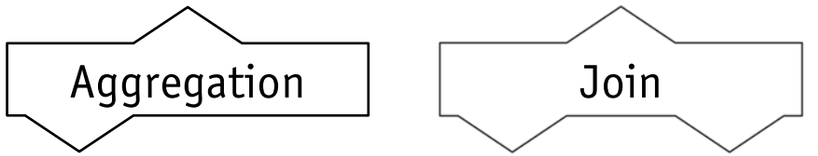
1. conceptually, the evaluation is initiated from the root, i.e., node A ⋈ B produces tuples as and when they are requested by its parent
2. when the root node gets a page of tuples from its left child (outer table), all matching inner tuples are retrieved (using either an index or a scan), and joined with matching outer tuples
3. the current page of outer tuples is then discarded, the next page is requested from the left child, and the process is repeated

- pipelined evaluation is thus a **control strategy** governing the rate at which different joins in the plan proceed
- results are produced, consumed, and discarded **one page at a time**, rather than writing intermediate result of joins to a temporary table

Bracket Model



Bracket Template



Operators

- Bracket Template
  - operating system and communication support
  - **central scheduler** can fit a single operator into the bracket on demand
- Operators
  - each operator **must** run its own thread, assumes all control within its thread, sends and receives data via networking procedures
  - essentially relies on networking procedures for pacing and flow control
  - each operator defines its own phases, synchronization points, and communication needs that must be known to the bracket implementation

Iterator Model

- To simplify the code to coordinate plan execution, all operators implement a **uniform** iterator interface
  - hides **internal implementation details** of each operators
  - assumes **pipelined evaluation** of the query plan (see next slide)

Iterator interface

open()
- initializes iterator by allocating input and output buffers
- used to pass parameters that modify operator behavior, e.g., a selection condition

next()
- calls **next** on each input operator
- executes operator-specific code to process input tuples
- places results in output buffer
- updates iterator state to keep track of how much input has been consumed

close()
- deallocates state information

Iterator Model

Exercise: pipelined vs. materialized evaluation in the iterator model

The iterator interface supports pipelining naturally, but what happens if the evaluation of an operator does not permit pipelining, e.g., blocking operators such as sort, aggregation, and certain joins?

- Iterators and indexes
  - iterator interface is also used to encapsulate access methods (indexes)
  - access methods are viewed as operators that produce a stream of tuples
  - **open()** is used to pass selection conditions that match the access path

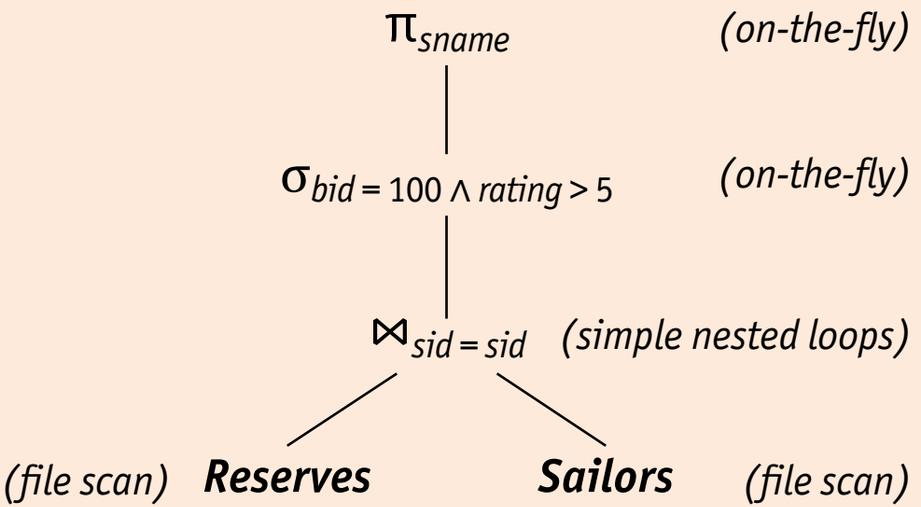
Iterator Model

Iterator	open()	next()	close()	State
print	open input	call **next()** on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call **next()** on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left *build* input, build hash table calling **next()** on build input, close build input, open right *probe* input	call **next()** in *probe* input until a match is found	close *probe* input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get **next()** item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling **next()** on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Motivating Alternative Plans

- Consider the cost of evaluating the plan shown on the right
  - **ignore** cost of writing out the result can as it is the same for all plans
  - cost of (simple nested loops) **join** is 1000 + 1000 · 500 page I/O operations
  - **selections** and **projection** are done on the fly and do not incur additional I/O operations

Running example: original plan



- **Total cost** of this plan is 501,000 page I/O operations
- Now, consider several alternative plans for evaluation this query
  - each plan improves on the original plan in a different way
  - cost benefit of each of these optimization is examined in detail

Pushing Selections

- Recall that joins are relatively expensive operations
  - sizes of the two inputs of the join determine its cost
  - a good **heuristic** is to reduce these inputs as much as possible

Pushing selections

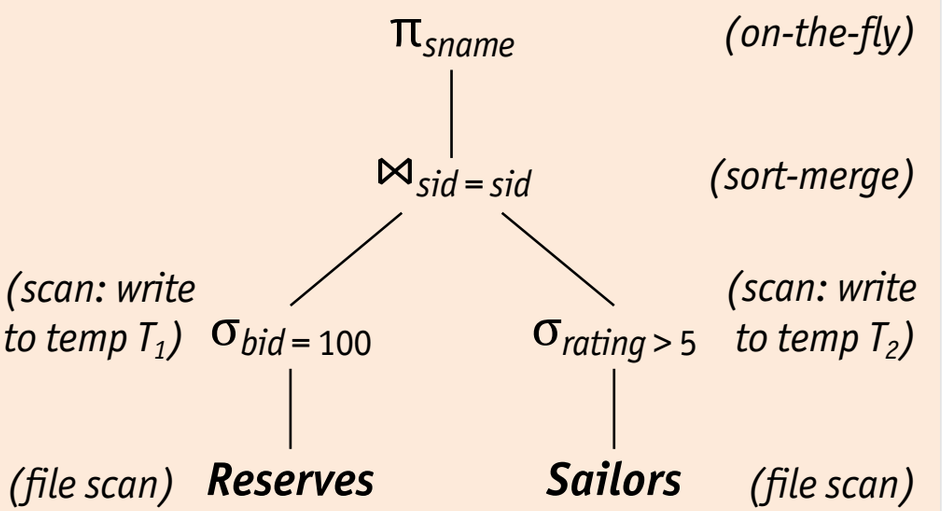
An example of this heuristic is to apply selections (σ) early: if a selection appears after a join, it is worth examining whether the selection can be **pushed** ahead of the join.

- Running example
  - selection *bid* = 100 only involves attributes of the ***Reserves*** table
  - selection *rating* > 5 only involves attributes of the ***Sailors*** table
  - both selections can be pushed and applied **before** the join

Pushing Selections

- Assumptions
  - selections performed using file scan and result written to temporary tables
  - sort-merge join used to join the temporary tables
  - five buffer pages are available to evaluate this plan

Running example: selections pushed



- Cost of selection $\sigma_{bid = 100}$
  - cost of scanning ***Reserves*** (1000 pages)
  - cost of writing result to table T_1 (this cost cannot be ignored)
- Additional information is needed to estimate size of T_1
  - **1 tuple**, if we assume a maximum of one reservation per boat
  - **10 pages**, if we know that there are 100 boats and we assume that reservations are uniformly distributed over boats (let's assume this!)

Pushing Selections

- Cost of selection $\sigma_{rating > 5}$
  - cost of scanning ***Sailors*** (500 pages)
  - cost of writing result to table T_2 (again, this cost cannot be ignored)
- Estimating the size of T_2
  - **250 pages**, if we assume ratings to be uniformly distributed over the range 1 to 10
- Cost of computing $\bowtie_{sid = sid}$ using a sort-merge join
  - use straightforward implementation: sort tables completely, then merge
  - using five available buffer pages, T_1 (10 pages) can be sorted in 2 passes (each pass reads and writes 10 pages), i.e., $2 \cdot 2 \cdot 10 =$ **40 page I/Os**
  - 4 passes needed to sort T_2 (250 pages), i.e., $2 \cdot 4 \cdot 250 =$ **2000 page I/Os**
  - to merge T_1 and T_2, both tables need to be scanned, i.e., the cost of this step is $10 + 250 =$ **260 page I/Os**
- Final projection π_{sname} is done on-the-fly and does not incur cost

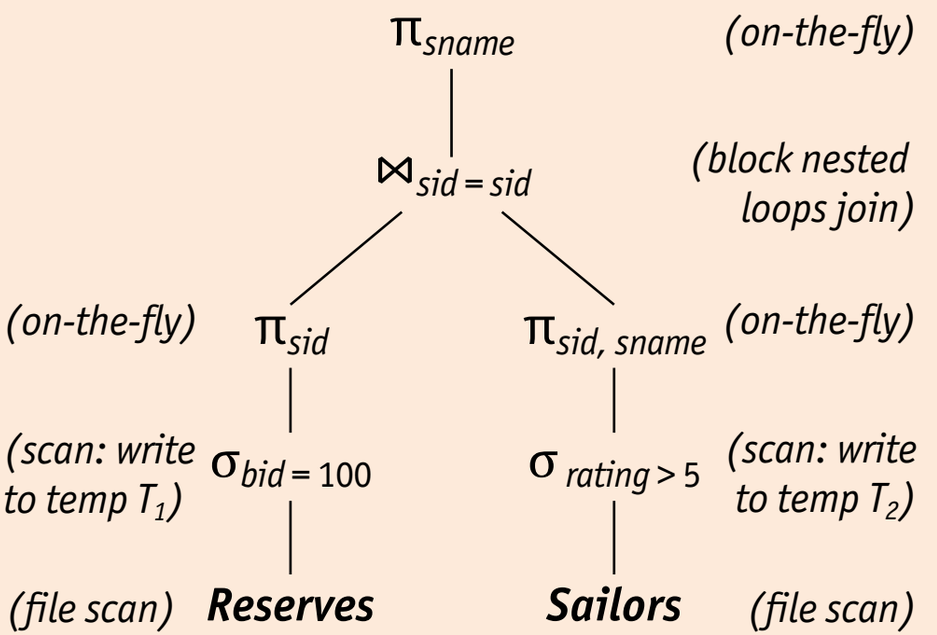
Pushing Selections

- **Total cost** of the plan is therefore 4060 page I/O operations
  - cost of **selections** is 1000 + 10 + 500 + 250 = 1760
  - cost of the **sort-merge join** is 40 + 2000 + 260 = 2300
- Assume that a **block nested loops join** is used instead
  - using T_1 as the outer table, for every three-page block of T_1, all of T_2 is scanned, i.e., T_2 is scanned for times (recall: five buffer pages available)
  - cost of **block nested loops join** is $10 + (4 \cdot 250) =$ **1010 page I/Os**
  - **total cost** of the plan is now 1760 + 1010 = 2770 page I/O operations
- Just like selections, pushing projections can also reduces the size of tables in a join
  - only *sid* of T_1 and ⟨*sid*, *sname*⟩ of T_2 are required to evaluate the query
  - as ***Reserves*** and ***Sailors*** are scanned, unnecessary attributes can be eliminated

Pushing and Introducing Projections

- **Pushing and introducing projections** is another heuristic to reduce the sizes of table in a join
  - only *sid* of T_1 and ⟨*sid*, *sname*⟩ of T_2 are required to evaluate the query
  - as T_1 and T_2 are written, unnecessary attributes can be eliminated
- Cost estimation
  - on-the-fly projections reduce the sizes of tables T_1 and T_2
  - reduction of T_1 substantial, since only integer attributes retained
  - T_1 now fits into three buffer pages
  - block nested loop join only needs to scan T_2 once (250 page I/Os)
- **Total cost** is 1760 + 250 = 2010 page I/O operations

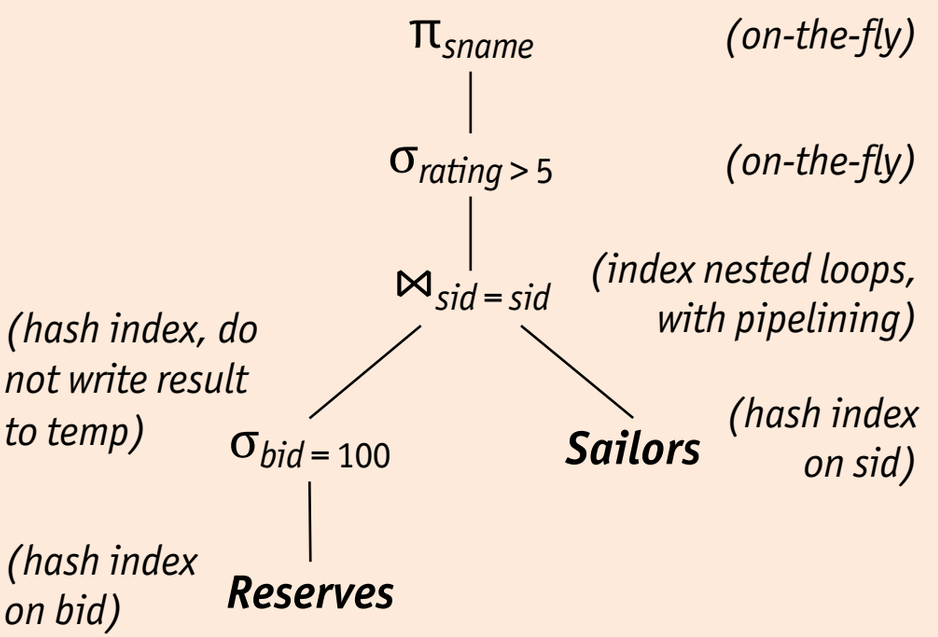
Running example: projections



Using Indexes

- If **indexes** are available on ***Reserves*** and ***Sailors***, even better query evaluation plans may be available
- Assumptions
  - clustered static hash index on ***Reserves***.*bid*
  - hash index on ***Sailors***.*sid*
- Cost of selection $\sigma_{bid = 100}$
  - **10 page I/Os**, if we make the same assumption as before
  - 100,000/100 = 1000 tuples are estimated to be selected
  - since index is clustered, the 1000 tuples are in 10 consecutive pages

Running example: using indexes



Using Indexes

- Computing $\bowtie_{sid = sid}$ using a index nested loop join
  - for each selected ***Reserves*** tuple, matching ***Sailors*** tuples are retrieved using the hash index on *sid*
  - selected ***Reserves*** tuples are not materialized and the join is pipelined
  - selection $\sigma_{rating > 5}$ and projection π_{sname} are performed on-the-fly
- Important points to note
  - since plan is fully pipelined, introduction projections is not needed
  - since *sid* is a key of ***Sailors***, at most one ***Sailors*** tuples matches a given ***Reserves*** tuple, therefore the cost of retrieving the matching tuple does not depend on whether the index on ***Sailors*** is clustered or not
  - selection $\sigma_{rating > 5}$ has not been pushed because it would require a scan of ***Sailors*** (assuming there is no index on *rating*) and because there is no index available on *sid* **after** the selection is applied

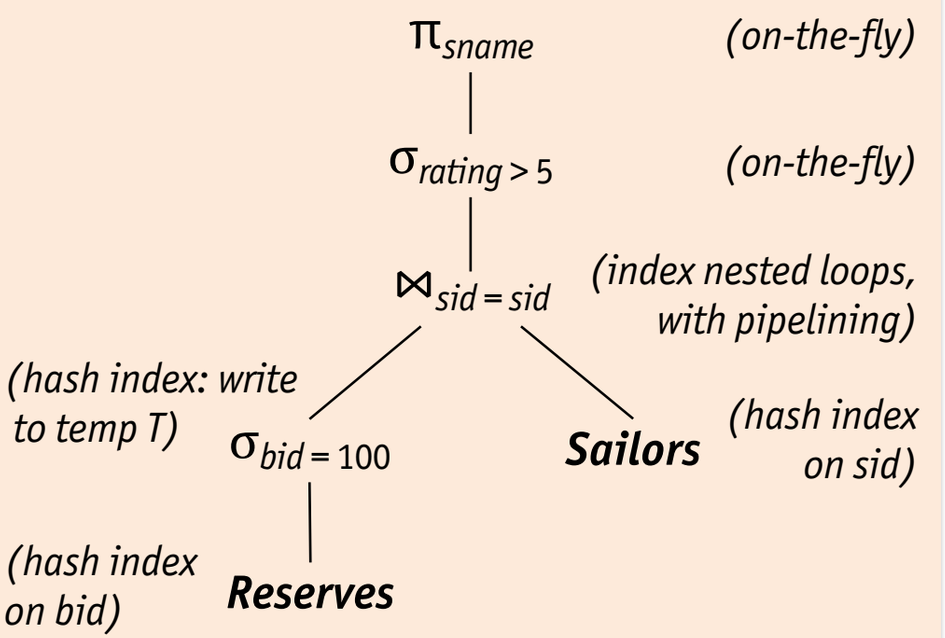
Using Indexes

- Cost of computing $\bowtie_{sid = sid}$ using an index nested loops join
  - for each of the 1000 ***Reserves*** tuples, the ***Sailors*** index on *sid* is probed
  - cost of probing ***Sailors*** index depends on whether the hash directory fits in memory and on the presence of overflow pages
  - assuming that the index uses variant ❶ for entries, **1.2 page I/Os** are a good estimate (if variant ❷ or ❸ is used the cost would be 2.2 page I/Os)
  - cost of **index nested loops join** is $1.2 \cdot 1000 = 1200$ page I/O operations
- **Total cost** is $10 + 1200 = \underline{1210}$ page I/O operations

Using Indexes

- If ***Sailors*** index on *sid* is clustered, plan can be further refined
- Assumption
  - materialize result of $\sigma_{bid = 100}$ to temporary table *T* and sort *T*
- Cost of selection $\sigma_{bid = 100}$
  - *T* has again 10 pages, selecting the tuples costs **10 page I/Os**
  - writing out the result costs another **10 page I/Os**
  - sorting with five buffer pages costs $2 \cdot 2 \cdot 10 =$ **40 page I/Os**
- Pipelining is **not always** best
  - Selected ***Reserve*** tuples can now be retrieved in order by *sid*
  - If a sailor has reserved the same boat many times, the matching ***Sailors*** tuple will be found in the buffer pool (on all but the first request)

Running example: using indexes



Using Indexes

- Combining pushing of selections with index use is very powerful
  - join operation may become trivial, if selected tuples from outer table join with a single inner tuple
  - performance gains with respect to naïve plan may be dramatic

Example: extended SPJ query

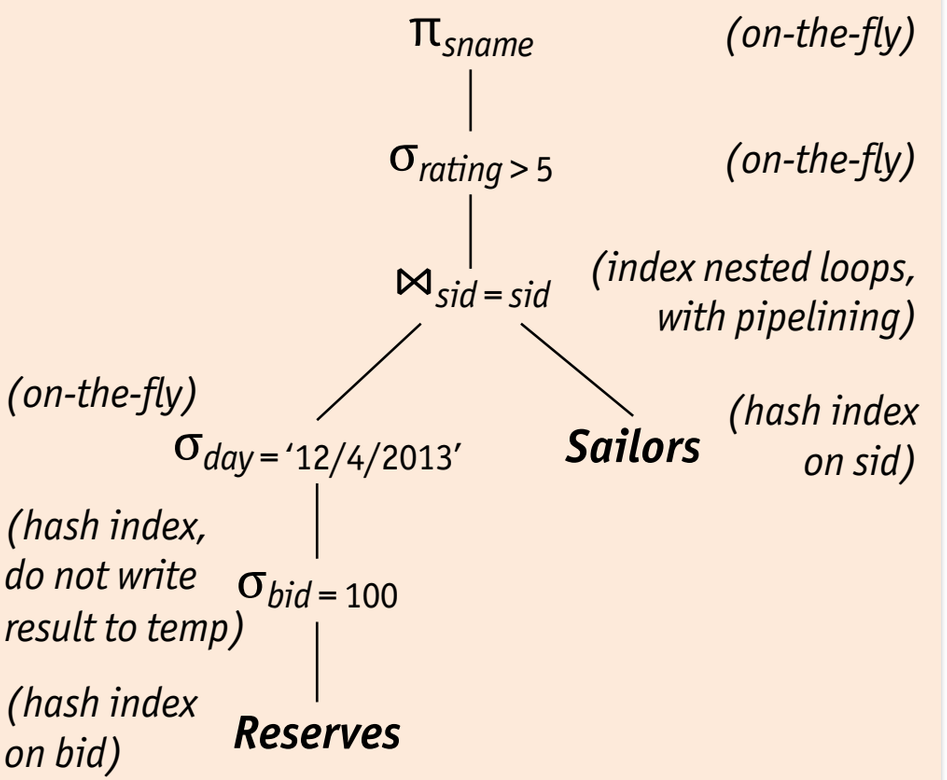
```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND R.bid = 100
       AND S.rating > 5 AND R.day = '12/04/2013'
```

- Remarks
  - extended query introduces additional selection $\sigma_{day = \text{'12/4/2013'}}$
  - assume that *bid* and *day* form a key on ***Reserves***

Using Indexes

- Selection $\sigma_{day = '12/4/2013'}$ is applied on-the-fly to the result of the selection $\sigma_{bid = 100}$ on the ***Reserves*** table
- Cost of selections
  - as before, cost of $\sigma_{bid = 100}$ is **10 page I/Os**
  - $\sigma_{day = '12/4/2013'}$ does not incur any cost as it is applied on-the-fly
- Cost of join
  - since ⟨*bid*, *day*⟩ is a key, there is only one outer tuple
  - cost of retrieving inner tuple is **1.2 page I/Os**
- **Total cost** is ~11 page I/Os
  - even with this selection, cost of naïve plan is still 501,000 I/Os!

Running example: using indexes



A Typical Query Optimizer

- Recall that a typical query optimizer performs **two main** tasks
  1. enumeration of plans
  2. cost estimation for each plan
- Plan enumeration
  - **logical level:** algebraic equivalences are used to identify equivalent expression for a given query
  - **physical level:** for each such equivalent expression, all available implementation techniques are considered for the operators involved
- Cost estimation
  - cost of each query evaluation plan is estimated (see previous slides)
  - plan with lowest estimated cost is chosen and executed

Plan Enumeration

Definition: equivalent expressions

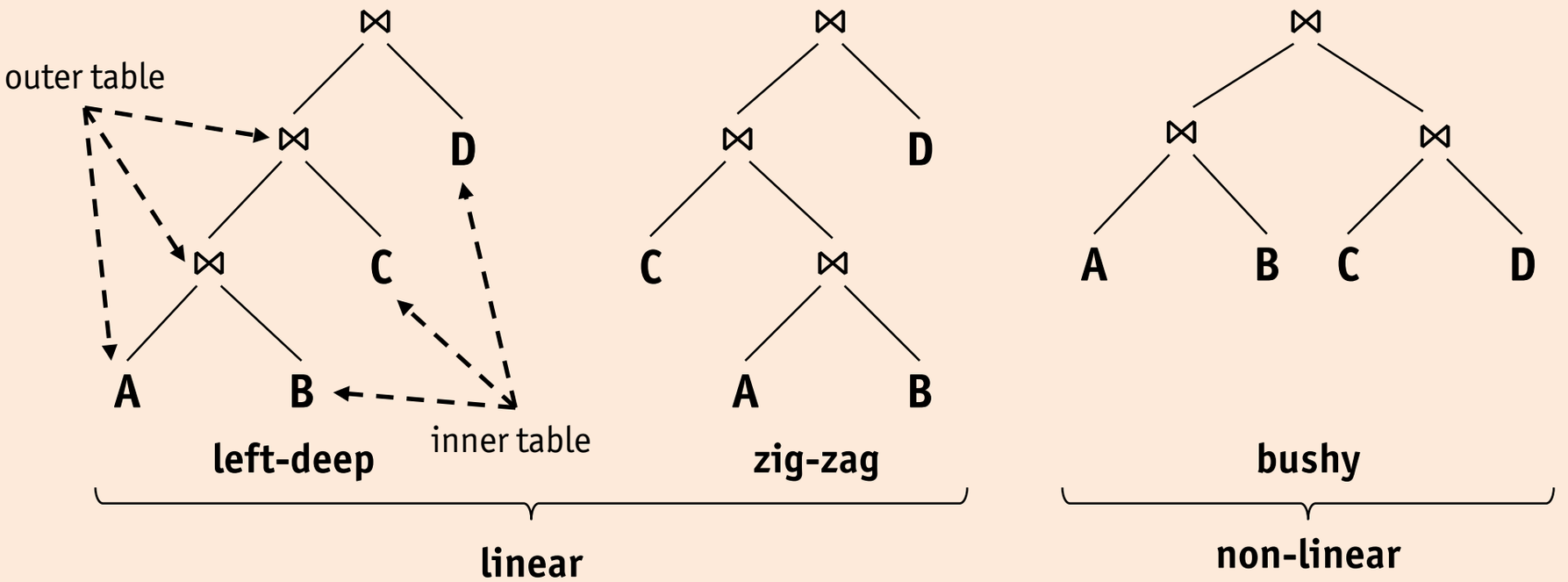
Two relational algebra expressions over the same set of input tables are said to be **equivalent** if the produce the same result on all instances of the input tables.

1. Transform SQL query into relational algebra
   - **FROM** $t_1, t_2, \ldots, t_n$ → $((t_1 \times t_2) \times \ldots) \times t_n$
   - **WHERE** a = v **AND** ... → $\sigma_{a=v \wedge \ldots}(\cdot)$
   - **SELECT** a, b, c → $\pi_{\{a,b,c\}}(\cdot)$
2. Apply relational algebra equivalences to convert initial expression into equivalent expression
   - **selections** (σ) and **cross-products** ($\times$) can be combined into **joins** ($\bowtie$)
   - **selections** (σ) and **projections** (π) can be "pushed" (or introduced) ahead of **joins** ($\bowtie$) to reduce the size of their inputs
   - **joins** ($\bowtie$) can be extensively reordered

Join Reordering

Example: three join trees

Consider the natural join of four tables, i.e., **A** ⋈ **B** ⋈ **C** ⋈ **D**. Based on relational algebra equivalences (joins commute), the three trees below are equivalent.



Exercise: enumerating join orders

How many left-deep trees for a natural join of four tables are there?

Left-Deep Plans

- Left-deep plans are typically the only query plans considered
  - **dynamic programming** used to efficiently search this class of plans
  - given a specific join order, selection and projection conditions are applied **as early as possible**
- This decision implies that the query optimizer will **not** find the best plan, if the best plan is not a left-deep plan!
- Reasons to concentrate on left-deep plans
  - it is necessary to **prune** the search space since the number of alternative plans increases rapidly with the number of joins in a query
  - left-deep plans can be translated into **fully pipelined** plans, in which all joins are evaluated using pipelining (a join result cannot be used as inner table in a pipelined plan, since inner tables must always be materialized)

Cost Estimation of a Plan

Definition: cost of a plan

The cost of a plan is the **sum** of the cost for the operators it contains. The cost of individual operators is estimated using information obtained from the system catalog.

- Cost of a plan in terms of I/O costs can be broken down
  1. **reading input tables** (for some join and sort algorithms, multiple times)
  2. **writing intermediate tables**
  3. **sorting the final result** (for duplicate elimination or output order)
- Remarks
  - unless one of the plans happens to produce output in the required order, the third part is common to all plans
  - in the common case that a fully pipelined plan is chosen, no intermediate tables are written (and read)

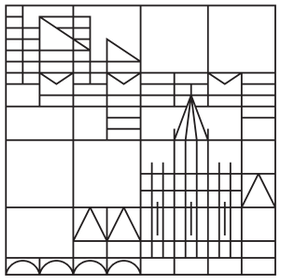
Cost Estimation of a Plan

- Cost of fully pipelined plans is dominated by reading input tables
  - greatly depends on the access paths used to read input tables
  - access paths that are repeatedly used in joins are especially important
- Not fully pipelined plans
  - cost of materializing results as temporary tables can be significant
  - estimation of materializing intermediate results based on its size
  - result size also influences the cost of the operator for which it is an input
- Operator result size
  - **σ** estimated by multiplying with reduction factor (of selection condition)
  - **π** is equal to input size (no duplicate elimination)
  - **⋈** estimated by multiplying maximum result size (product of sizes of input tables) with reduction factor (of join condition)

Cost Estimation of a Plan

Exercise: approximating the reduction factor of a join

Suppose a query optimizer needs to estimate the result size of $\boldsymbol{A} \bowtie_{aid=bid} \boldsymbol{B}$. Further assume that there are indexes I_1 and I_2 on *aid* and *bid*, respectively. How would you approximate the reduction factor of the join condition?



TO BE CONTINUED...