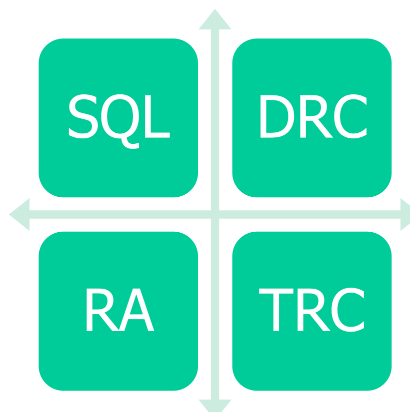
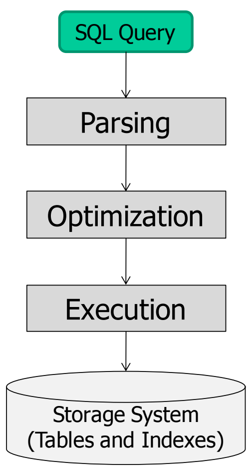
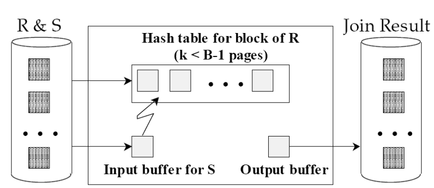
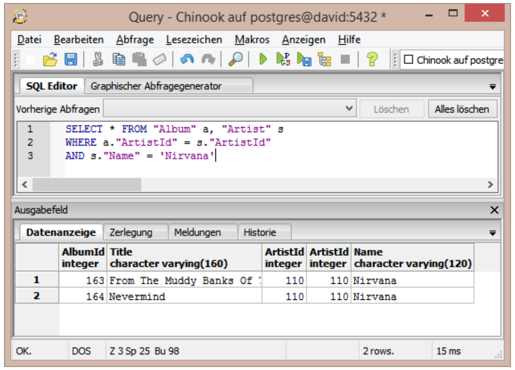
Chapter 2

Query Languages and Implementation of Relational Operators









Contents

Running Example

DB Schema:

DEPT
dno
dname
mgr

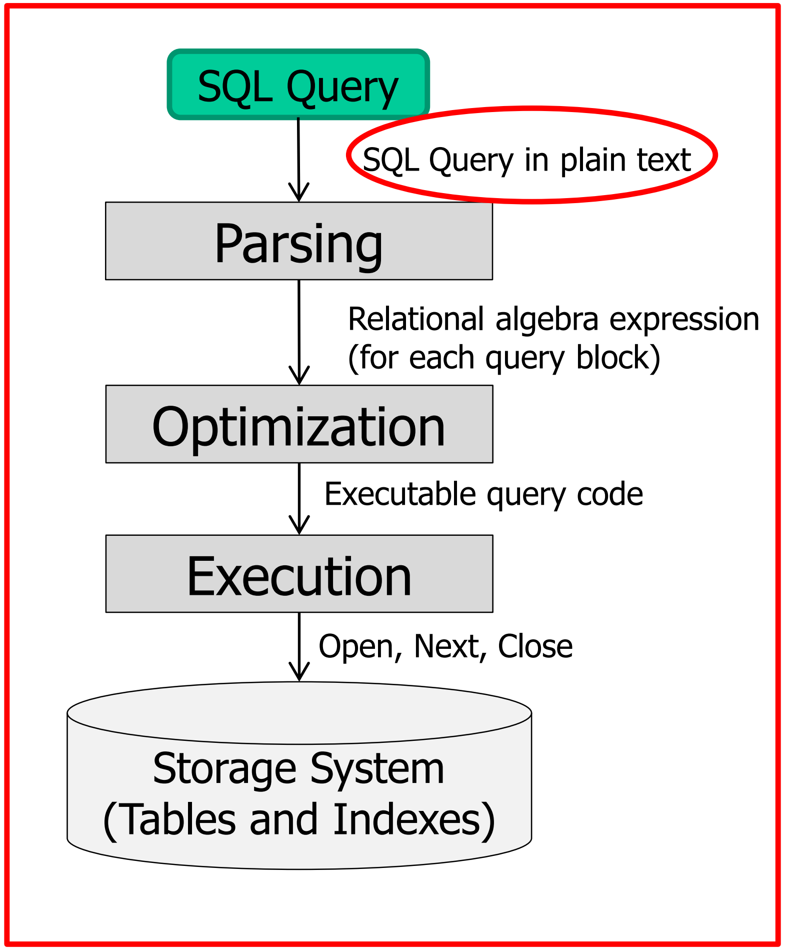
EMPL
eno
name
marstat
salary
dno

OFFICE
floor
room
eno

Integrity constraints:

- Inter-relational dependencies (foreign keys)
  - EMPL[dno] $\subseteq$ DEPT[dno]
  - OFFICE[eno] $\subseteq$ EMPL[eno]
  - DEPT[mgr] $\subseteq$ EMPL[eno]
- Functional dependencies
  - EMPL: name $\rightarrow$ eno
  - DEPT: mgr $\rightarrow$ dno
- Value constraints:
  - EMPL: 10,000 < salary < 90,000

Query: Names of single employees in computer department who make less than 40,000

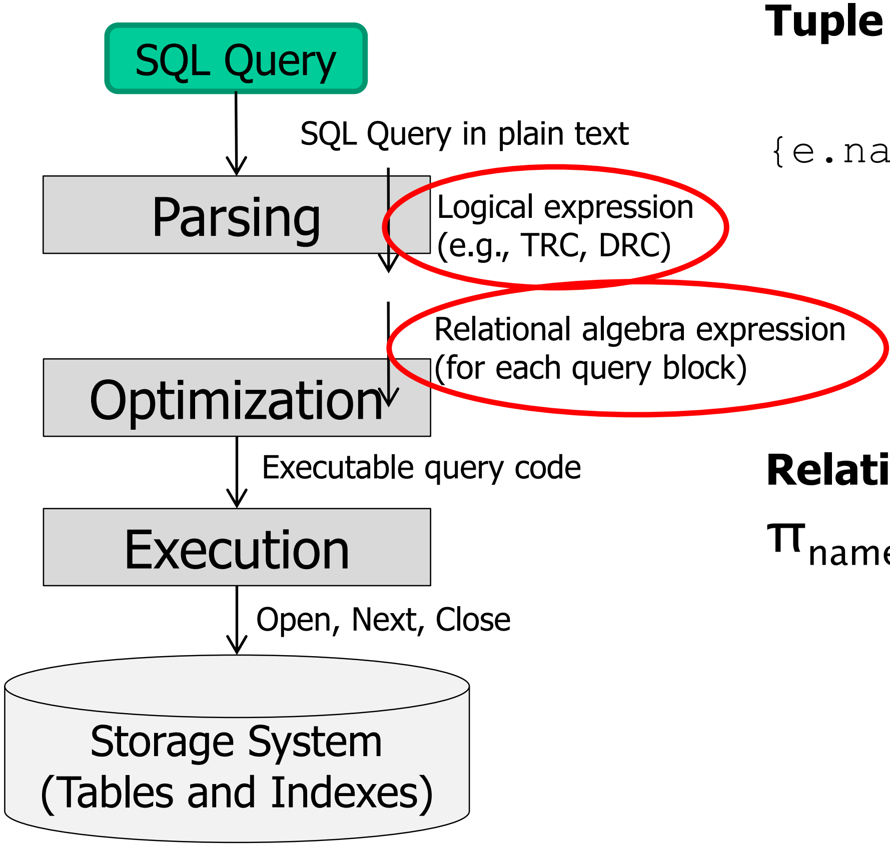


SQL (declarative):

```
SELECT
   e.name FROM EMPL e, DEPT d
WHERE
   e.salary < 40,000 AND
   e.marstat = 'single' AND
   d.dname = 'computer' AND
   d.dno = e.dno
```

Query Processing Chain (2)

Query: Names of single employees in computer department who make less than 40,000



Tuple Relational Calculus (TRC, declarative):

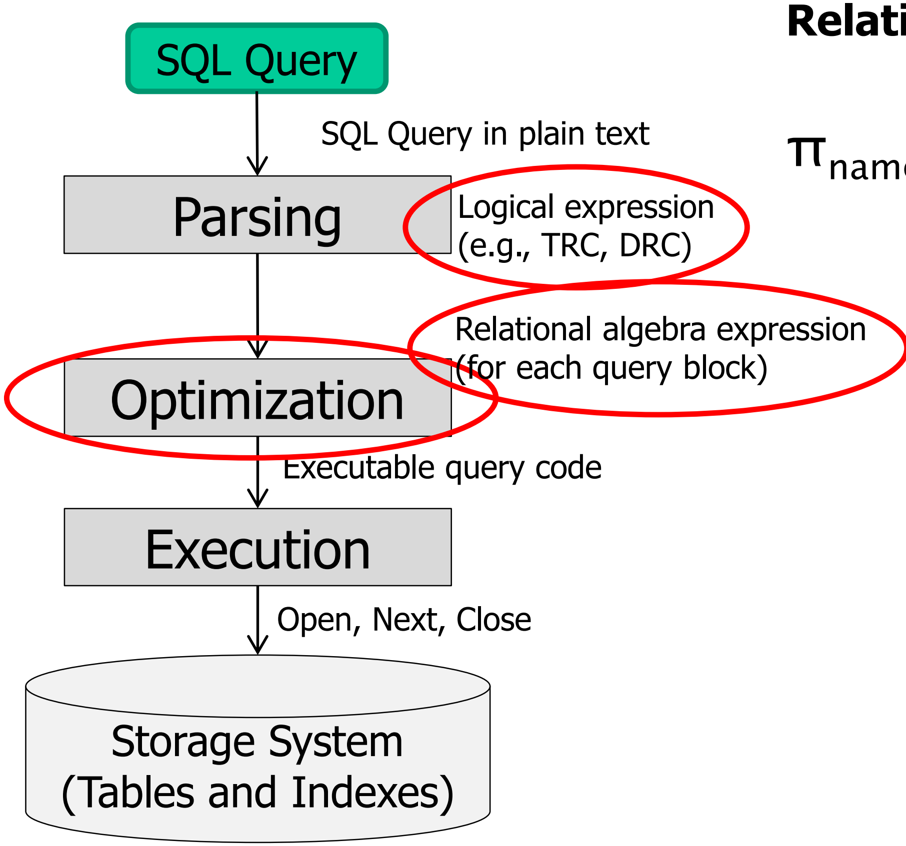
```
{e.name | e ∈ EMPL ∧ ∃d ∈ DEPT ∧
      e.salary<40,000 ∧
      e.marstat='single' ∧
      d.dname='computer' ∧ d.dno=e.dno}
```

Relational Algebra (RA, procedural)

$$\pi_{name}(\sigma_{dname='computer' \wedge salary<40{,}000 \wedge marstat='single'}(EMPL \bowtie DEPT))$$

Query Processing Chain (3)

Query: Names of single employees in computer department who make less than 40,000



Relational Algebra (optimized):

$$\pi_{name}(\sigma_{salary<40,000 \wedge marstat='single'}(EMPL) \bowtie \sigma_{dname='computer'}(DEPT))$$

Query Evaluation and Optimization

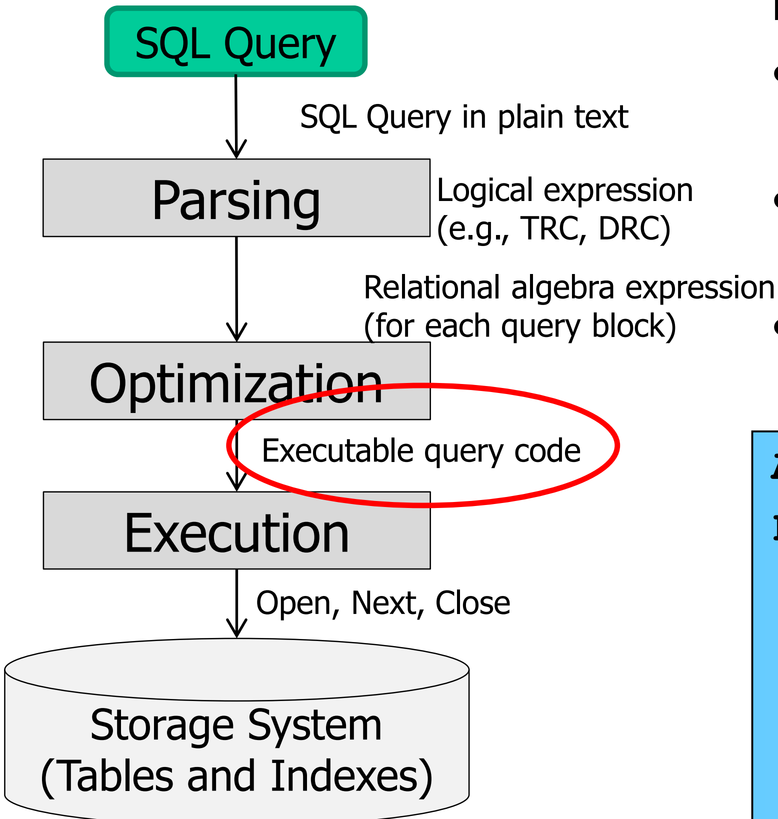
- Large number of optimization strategies possible. Reduction by
  - Syntactic query transformation
  - Semantic query transformation

- (Implicit) enumeration and evaluation of remaining strategies:
  - Definition of the space of data structures and operations
  - Effects of operations on the size of intermediate results
  - Effects of the size of intermediate results and operations on communication costs, storage costs and CPU costs

- Query support and query optimization by “investments”:
  - Sorting
  - Index access
  - Access paths
  - Partial operation evaluation

Query Processing Chain (3)

Query: Names of single employees in computer department who make less than 40,000



Nested Loop Join

- Iterate over both relations using two nested loops
- Check join and selection conditions in inner loop
- If ok, add projected attribute(s) to result set

```
ANSWER:=[];
FOR EACH e in EMPL DO
  FOR EACH d IN DEPT DO
    IF  e.salary<40,000 AND
        e.marstat='single' AND
        d.dname='computer' AND
        d.dno=e.dno
    THEN ANSWER:+[<e.name>];
```

Improved Nested Loop Join

Heuristics to improve query execution: Selection before join!

Improved Nested Loop Join

1. Scan one relation, check selection conditions, put result into temporary buffer
2. Scan second relation, check join & selection condition using intermediate result from temporary buffer, create result set

```
DNOLIST:=[];
FOR EACH d IN DEPT DO
  IF d.dname='computer' THEN DNOLIST:+[<d.dno>];
ANSWER:=[];
FOR EACH e in EMPL DO
IF  e.salary<40,000 AND e.marstat='single'
    THEN FOR EACH d IN DNOLIST DO
      IF d.dno=e.dno THEN ANSWER:+[<e.name>];
```

Summary

- Queries can be represented in various languages
- Query evaluation requires the transformation of queries from a user-friendly query-language (e.g., SQL) to an implementation-oriented language (e.g., RA)
- Query optimization can be applied along this transformation process (e.g., syntactical and semantical optimizations), but the choice of operators and access plans is significant for the query performance
- Kind of operator implementation also can make a difference

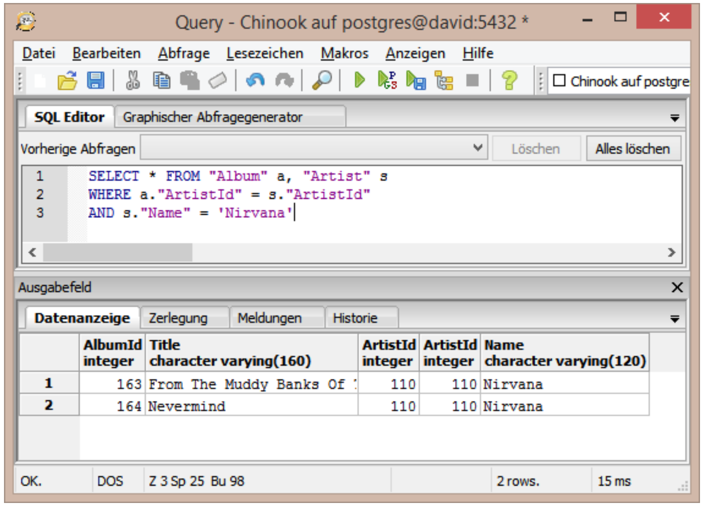
2.1 Query Languages in a Nutshell

Requirements concerning query representation:

- **Usability:**
  The representation should be appealing and comprehensible for the user
- **Expressiveness vs. Complexity :**
  Formulation of desired queries should be possible
  (Standard: the "Relational Completeness")
- **Formal manipulability, commutability, parallelism and pipelining**

2.1.1 SQL

- Structured Query Language
- Standard language for most relational DBMS
  - Standardized since 1986, current version ISO/IEC 9075:2011
- Sublanguages for definition of schema, data, transactions, access rights, and queries
- Very similar to TRC as variables represent tuples, but SQL has also constructs from RA (e.g., JOIN)



Structure of SQL Queries

Clause	**Logical** Order	Semantics

`SELECT`
 	
`(DISTINCT)`
	5	**Projection**: Only the selected columns will be in the result; apply functions (`sum`, `avg`, ...)
		
		Remove duplicates from result

FROM	1	Compute **Cartesian Product** (... **,** ...) or **Join** (... `JOIN` ... `ON` ...) over the given tables
WHERE	2	**Selection**: select all tuples satisfying the WHERE condition
GROUP BY	3	**Group** tuples
HAVING	4	Select only those **group**ed tuples which satisfy the **HAVING-Condition**
ORDER BY	6	**Sort** the result

2.1.2 Relational Algebra

- **Selection:** $\sigma_{dname=`computer'}(DEPT)$ $\equiv$ SELECT * FROM DEPT d WHERE d.dname=‘computer’
- **Projection:** $\pi_{name}(EMPL)$ $\equiv$ SELECT e.name FROM EMPL e
- **Join:** $EMPL \bowtie DEPT$ $\equiv$ SELECT * FROM EMPL NATURAL JOIN DEPT
  $\equiv$ SELECT * FROM EMPL e, DEPT d WHERE e.dno=d.dno
- **Union:** $EMPL1 \cup EMPL2$ $\equiv$ SELECT * FROM EMPL1 UNION SELECT * FROM EMPL2
- **Rename Relation:** $\rho_{boss}(EMPL)$ $\equiv$ SELECT boss.* FROM EMPL boss
- **Rename Attribute:** $\rho_{(n \leftarrow name)}(EMPL)$ $\equiv$ SELECT e.name AS n FROM EMPL e

Analogous: Difference (-), Intersection ($\cap$), Cartesian Product (x)

Semi-join

$$\text{rel}_1 \ltimes_{a=b} \text{rel}_2 = \pi_{att\_rel_1}(rel_1 \bowtie_{a=b} rel_2)$$

- Reducing operation, i.e. $|\text{rel}_1 \ltimes_{a=b} \text{rel}_2| \leq |\text{rel}_1|$
- Example:

    Employees who are managers of a department:

$\text{EMPL} \ltimes_{eno=mgr} \text{DEPT} \equiv$

eno	name	marstat	salary	dno	dno	dname	mgr
1	Jarke	1	xxxxx	5	5	DBIS	Jarke
2	Seidl	1	xxxxx	9	9	DMDE	Seidl

```
SELECT e.eno, e.name,
       e.marstat, e.salary, e.dno
FROM   EMPL e, DEPT d
WHERE e.dno = d.dno
```

eno	name	marstat	salary	dno
1	Jarke	1	xxxxx	5
2	Seidl	1	xxxxx	9

2.1.3 Tuple Relational Calculus (TRC)

Example Query:

“Names of single computer people who make less than 40,000€ ?”

The tuple calculus is a first-order predicate calculus.

Goal list

Free variable

Monadic term: Selection or Restriction

Dyadic term: Join condition

Quantified subquery

$$\{\langle e.name\rangle \mid e \in EMPL:\ e.salary<40{,}000 \wedge e.marstat='single' \wedge \exists\, \underbrace{d \in DEPT(}_{\text{Range query}} d.dno=e.dno \wedge d.dname='computer')\}$$

Existence Quantor

Range query

Queries in TRC

A *query* in the Tuple Relational Calculus (TRC) is a relation-valued expression of the form:

$$\{[r_1.A_1,\ldots,r_n.A_n] \mid r_1 \in R_1,\ldots, r_n \in R_n: \Phi\}$$

Φ is a formula in TRC, in which $r_1, \ldots, r_n$ are the only free tuple variables.

Atomic formulas in TRC

- **Range Expression:** $t \in R$, where t is a tuple variable and R a relation
- **Dyadic Term:** *t.A op u.B,* where t,u are tuple variables, A,B are attributes of t and u, and op is a comparison operator (=,<>,<=,<,>,>=)
- **Monadic Term:** *t.A op c,* where t is a tuple variable, A an attribute of t, c a constant, and op is a comparison operator (=,<>,<=,<,>,>=)
- **TRUE** and **FALSE**

Formulas in TRC

- All atomic formulas are formulas in TRC.

- If Φ, Ψ are formulas in TRC, then

$\Phi \wedge \Psi$	(Conjunction)
$\Phi \vee \Psi$	(Disjunction)
$\neg\Phi$	(Negation)
(Φ)	are also formulas in TRC.

- If Φ is a formula in TRC, and t is a free tuple variable in Φ, then

$\exists t\ \Phi$	(Existential Quantification)
$\forall t\ \Phi$	(Universal Quantification)

  are also formulas in TRC. Φ is called the *scope* of t.

- There are no other formulas in TRC.

Examples

- Names of departments with all employees earning less than 40,000.

- Employees earning less than 40,000 and working on the same floor as their boss.

TRC vs. SQL

$\{\langle e.name\rangle \mid$
$e \in EMPL \wedge \exists\, d \in DEPT$
$e.salary<40{,}000 \wedge$
$e.marstat='single' \wedge$
$d.dno=e.dno \wedge$
$d.dname='computer' \}$

SELECT e.name
FROM EMPL e, DEPT d
WHERE
e.salary<40,000 **AND**
e.marstat='single' **AND**
d.dno=e.dno **AND**
d.dname='computer'

TRC vs. SQL

All employees with name ´Müller´, who did not publish papers in 2011 or currently teach undergraduate lectures.

$$\{e \mid e \in EMPL$$
$$\wedge\ e.name = \text{´Müller´}\ \wedge$$
$$(\neg($$
$$\exists p \in PAPERS$$
$$(p.year = 2011 \wedge p.eno = e.eno)) \vee$$
$$\exists c \in COURSES \wedge \exists\ t \in TIMETABLE$$
$$(c.level = \text{´Bachelor´}\ \wedge$$
$$(t.cno = c.cno \wedge t.eno = e.eno)))\ \}$$

```
SELECT e.* FROM EMPL e
WHERE
e.name='Müller' AND
((NOT EXISTS
    (SELECT p.*
     FROM PAPERS p
     WHERE
          p.year=2011 AND
          p.eno=e.eno)) OR
          EXISTS (SELECT c.*
          FROM COURSES c, TIMETABLE t
          WHERE c.level='Bachelor' AND
          t.cno=c.cno AND t.eno=e.eno))
```

2.1.4 Domain Relational Calculus (DRC)

- **Domain variables** $x_i \in DOM_i$ represent attributes

- **Atomic formulas:**
  1. $R(x_1,x_2,...,x_k)$ for a k-ary relation R and x_i are either constants or domain variables
  2. $x_i \; \theta \; x_j$ with $\theta \in \{=, <, \leq, >, \geq, \neq\}$, x_i, x_j are either constants or domain variables

- **Formulas:**
  1. Atomic formulas
  2. If A, B are formulas, then also
     $\neg A$
     $A \wedge B$
     $A \vee B$ are formulas in DRC
  3. If A is a formula and x is a free variable, then also
     $\exists x(A)$
     $\forall x(A)$ are formulas in DRC.

Queries in DRC

A *query* in the Domain Relational Calculus (DRC) is a relation-valued expression of the form:

$$\{[x_1,\ldots,x_n] \mid \Phi\}$$

Φ is a formula in DRC, in which $x_1, \ldots, x_n$ are the only free variables.

Example:

‘Names of single computer scientists with salary less than 40,000‘

Goal List

Domain Variables

{name | ∃eno,marstat,salary,dno,dname,mgr
EMPL(eno,name,marstat,salary,dno) ∧
DEPT(dno,dname,mgr) ∧
marstat='single' ∧ dname='computer' ∧ salary<40,000 }

Relation

Important Observations

- Formulas in DRC (and TRC) are called *safe*, if they describe a *finite* result set.
- The expressive power of safe DRC is equivalent to safe TRC and to Relational Algebra.
- DRC can also be used to describe facts and rules (e.g. PROLOG/DATALOG, see section “Deductive Databases”)
- Tableaux representing DRC queries are suited particularly for query simplification (see Chapter “Query Optimization”)

2.1.5 Expressiveness vs. Complexity (1)

Expressiveness

[Chandra & Harel, 1980]

- Tableaux queries
- Conjunctive queries
- Relational complete queries (RA, TRC, DRC)
- Fixpoint/Horn-clause queries
- First-Order Predicate Calculus (with disjunction / negation)
- Computable queries

Expressiveness increases, but also complexity!

2.1.5 Expressiveness vs. Complexity (2)

Complexity

[Chandra & Harel, 1980]

1. Data complexity (depends on DB size)
   $\Rightarrow$ polynomial only until fixpoint queries
   Conclusion: Larger DBs are problematic for more complex queries

2. Expression complexity (depends on query length)
   – Computation of result in polynomial time, e.g. "tree"-structured queries (semi-join queries)
   – Optimization of expressions in polynomial time, e.g. "simple tableaux queries"

The expression complexity is generally higher than the data complexity.

$\Rightarrow$ DB are "simpler" than knowledge bases.

Summary of 2.1

- There are many query languages to specify a query in a relational database
  - Declarative: SQL, TRC, DRC
  - Procedural: Relational Algebra

- Basis: Relational completeness
  - A query language is relationally complete if it is at least as expressive as relational algebra.

- Logical transformations and reasoning can be applied to queries in TRC or DRC to achieve "simpler" queries (➔ "Query Optimization")

2.2 Implementation of Relational Operators

2.2.1 Sorting

2.2.2 Selection

2.2.3 Projection

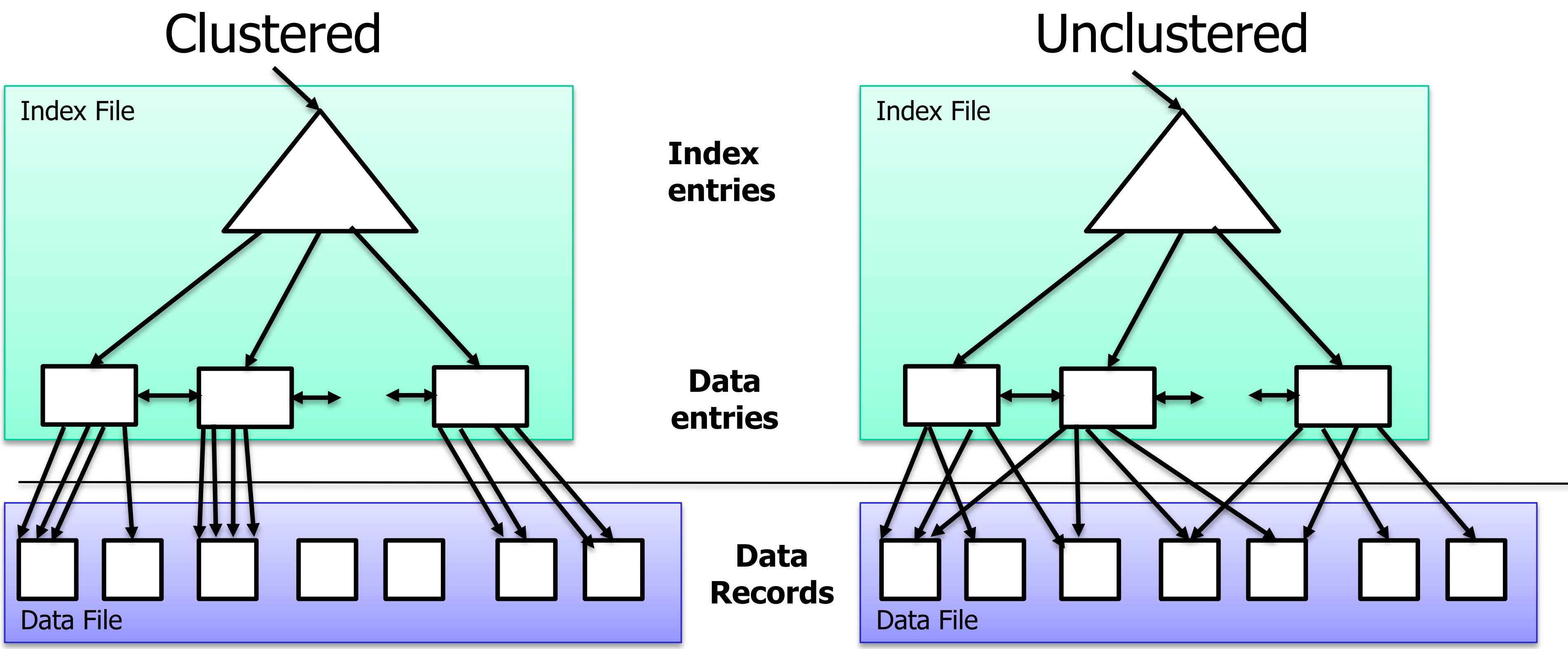
2.2.4 Join

2.2.5 Set Operations: Union and Difference

2.2.6 Aggregate Operations

Indexing

- Data structure organizing data records on disk to optimize data access times
- **Primary index:** on set of attributes including primary key
- **Secondary index:** on set of attributes excluding primary key
- **Clustered vs. Unclustered:** Data records in data file are sorted according index



[Ramakrishnan & Gehrke, 2003]

I/O Costs of Access Paths

[Ramakrishnan & Gehrke, 2003]

	Scan of Relation	Equality Selection	Range Selection	Insert	Delete
Heap File	B	0.5 B	B	2	0.5 B + 1
Sorted File	B	$\log_2 B$	$\log_2 B$ + #matching pages	$\log_2 B$ + B	$\log_2 B$ + B
Clustered Tree Index*	0.15 B + 1.5 B	1+ $\log_G 0.15B$	$\log_G 0.15B$ + #matching pages	3+ $\log_G 0.15B$	3+ $\log_G 0.15B$
Unclustered Tree Index*	0.15 B + R·B	1+ $\log_G 0.15B$	$\log_G 0.15B$ + #matching records	3+ $\log_G 0.15B$	3+ $\log_G 0.15B$
Unclustered Hash Index**	B· (R+0.125)	2	B	4	4

- **B:** number of data pages
- **R:** Number of records per page
- **G:** Fan-out of tree index

*:data entry in leaf is 10% of record size, average load per page is 67% ➔ 0.15B leaf pages and 1.5 B pages in clustered heap file

**: avg. load per index page is 80%, 10% data entry ➔ 0.125B pages for data entries

Heap File

- Example for heap structure
  - Relation Dept(dno,dname,mgr)
- No order of records
- Max. 4 records per page
- Cost for most operations
  ➔ B: all pages have to be read

P1

1	Research	4711
2	Marketing	815
5	Finance	3923
30	Facility	1123

P2

4	Distribution	2391
99	CEO	9999
11	Data Mgmt	1112
33	Legal	7777

P3

12	Board	5432
14	Travel	1234
10	Science	4567
6	Software	4994

P4

18	Production	6789
17	Logistics	9876
19	HR	3456
21	Education	3675

P5

36	Catering	6666
7	IT	2132
8	Helpdesk	3043
9	Controlling	8485

P6

32	Order	9888
29	Building	6776
3	Sales	3319
23	Planning	2346

Sorted File

- Records sorted according to dno
- Binary search possible
  ➔ $\log_2 B$

P1

1	Research	4711
2	Marketing	815
3	Sales	3319
4	Distribution	2391

P2

5	Finance	3923
6	Software	4994
7	IT	2132
8	Helpdesk	3043

P3

9	Controlling	8485
10	Science	4567
11	Data Mgmt	1112
12	Board	5432

P4

14	Travel	1234
17	Logistics	9876
18	Production	6789
19	HR	3456

P5

21	Education	3675
23	Planning	2346
29	Building	6776
30	Facility	1123

P6

32	Order	9888
33	Legal	7777
36	Catering	6666
99	CEO	9999

Clustered Tree Index

- Index on DNO
- B* tree with k=6 and k*=3
- Each node is exactly one page
- Nodes are not full; thus, additional storage necessary
  1.5B instead of 1.0B
- Cost for most operations:
  height of the tree ($\log_G 0.15B$)
  +1 for data page

R

Key	Child
	B1
7	B2
14	B3
29	B4

B1

Key	Page	Offset
Prev	NULL	
1	P1	1
2	P1	2
3	P1	3
4	P2	1
5	P2	2
6	P2	4
Next	B2	

B2

Key	Page	Offset
Prev	B1	
7	P3	1
8	P3	2
9	P3	3
10	P4	1
11	P4	2
12	P4	3
Next	B3	

B3

Key	Page	Offset
Prev	B2	
14	P5	1
17	P5	2
18	P5	3
19	P6	1
21	P6	2
23	P6	3
Next	B4	

B4

Key	Page	Offset
Prev	B3	
29	P7	1
30	P7	2
32	P8	1
33	P8	2
36	P9	1
99	P9	3
Next	NULL	

P1

1	Research	4711
2	Marketing	815
3	Sales	3319

P2

4	Distribution	2391
5	Finance	3923
6	Software	4994

P3

7	IT	2132
8	Helpdesk	3043
9	Controlling	8485

P4

10	Science	4567
11	Data Mgmt	1112
12	Board	5432

P5

14	Travel	1234
17	Logistics	9876
18	Production	6789

P6

19	HR	3456
21	Education	3675
23	Planning	2346

P7

29	Building	6776
30	Facility	1123

P8

32	Order	9888
33	Legal	7777

P9

36	Catering	6666
99	CEO	9999

Unclustered Tree Index

- Index on MGR
- B* tree with k=6 and k*=3
- Each node is exactly one page
- Cost for most operations: height of the tree ($\log_G 0.15B$) + number of records

R

Key	Child
	B1
2355	B2
4223	B3
6789	B4

B1

Key	Page	Offset
Prev	NULL	
815	P1	2
1112	P4	2
1123	P7	2
1234	P5	1
2132	P3	1
2346	P6	3
Next	B2	

B2

Key	Page	Offset
Prev	B1	
2391	P2	1
3043	P3	2
3319	P1	3
3456	P6	1
3675	P6	2
3923	P2	2
Next	B3	

B3

Key	Page	Offset
Prev	B2	
4567	P4	1
4711	P1	1
4994	P2	4
5432	P4	3
6666	P9	1
6776	P7	1
Next	B4	

B4

Key	Page	Offset
Prev	B3	
6789	P5	3
7777	P8	2
8485	P3	3
9876	P5	2
9888	P8	1
9999	P9	3
Next	NULL	

P1

1	Research	4711
2	Marketing	815
3	Sales	3319

P2

4	Distribution	2391
5	Finance	3923
6	Software	4994

P3

7	IT	2132
8	Helpdesk	3043
9	Controlling	8485

P4

10	Science	4567
11	Data Mgmt	1112
12	Board	5432

P5

14	Travel	1234
17	Logistics	9876
18	Production	6789

P6

19	HR	3456
21	Education	3675
23	Planning	2346

P7

29	Building	6776
30	Facility	1123

P8

32	Order	9888
33	Legal	7777

P9

36	Catering	6666
99	CEO	9999

Unclustered Hash Index

- Index on DNAME
- Hash function h(x) = first letter
- 6 hash entries per page, but only 4 entries are used

- This hashing preserves the order, but typically the hash function does not maintain the order. Thus, range queries cannot be efficiently done with a hash index.

Hash	Key1	Offset1	Key2	Offset2	Key3	Offset3
A						
B	P4	3	P7	1		
C	P9	1	P9	3	P3	3
D	P4	2	P2	1		
E	P6	2				
F	P7	2	P2	2		
G						
H	P3	2	P6	1		
I	P3	1				
J						
K						
L	P8	2	P5	2		
M	P1	2				
N						
O	P8	1				
P	P6	3	P5	3		
Q						
R	P1	1				
S	P1	3	P4	1	P2	4
T	P5	1				
U						
V						
W						
X						
Y						
Z						

P1

1	Research	4711
2	Marketing	815
3	Sales	3319

P2

4	Distribution	2391
5	Finance	3923
6	Software	4994

P3

7	IT	2132
8	Helpdesk	3043
9	Controlling	8485

P4

10	Science	4567
11	Data Mgmt	1112
12	Board	5432

P5

14	Travel	1234
17	Logistics	9876
18	Production	6789

P6

19	HR	3456
21	Education	3675
23	Planning	2346

P7

29	Building	6776
30	Facility	1123

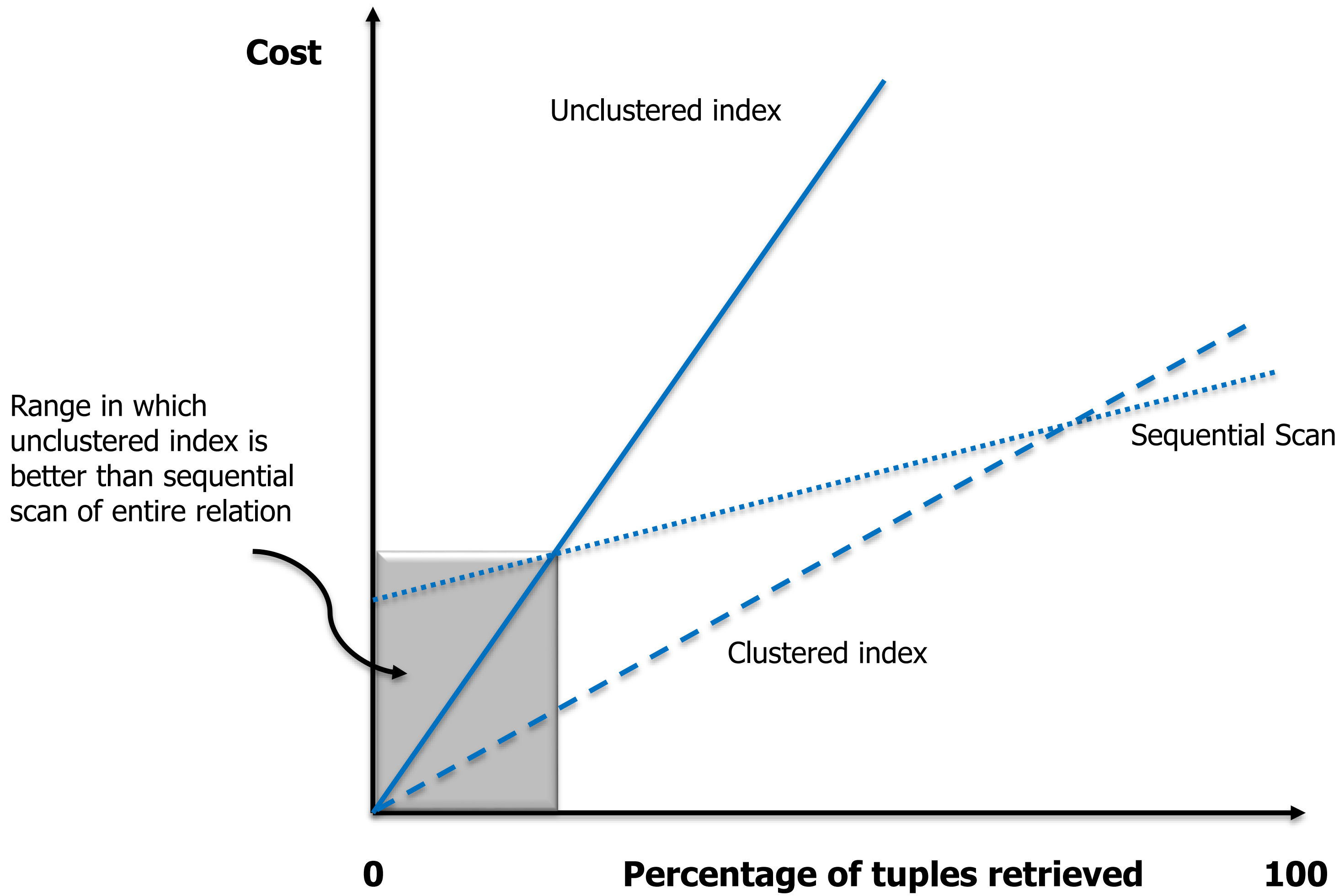
P8

32	Order	9888
33	Legal	7777

P9

36	Catering	6666
99	CEO	9999

Impact of Clustering



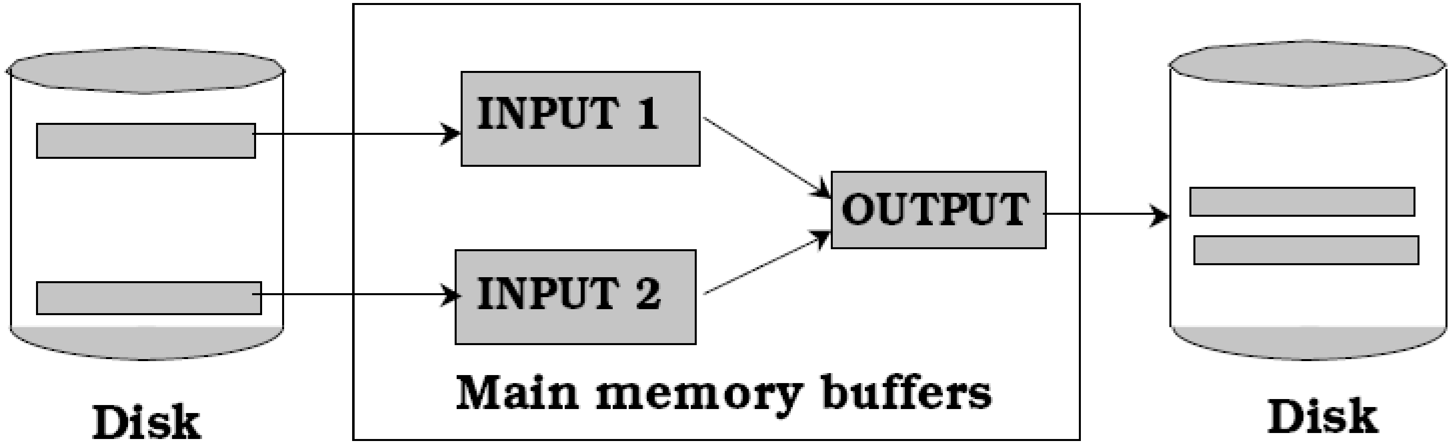
[Ramakrishnan & Gehrke, 2003]

2.2.1 Sorting

- Data requested in sorted order, e.g., find students in increasing age order
- Sorting is first step in bulk loading B+ tree index
- Sorting is useful for eliminating duplicate copies in a collection of records
- Sort-merge join algorithm involves sorting
- **Problem:** sort X GB of data with Y GB of RAM (X>>Y)

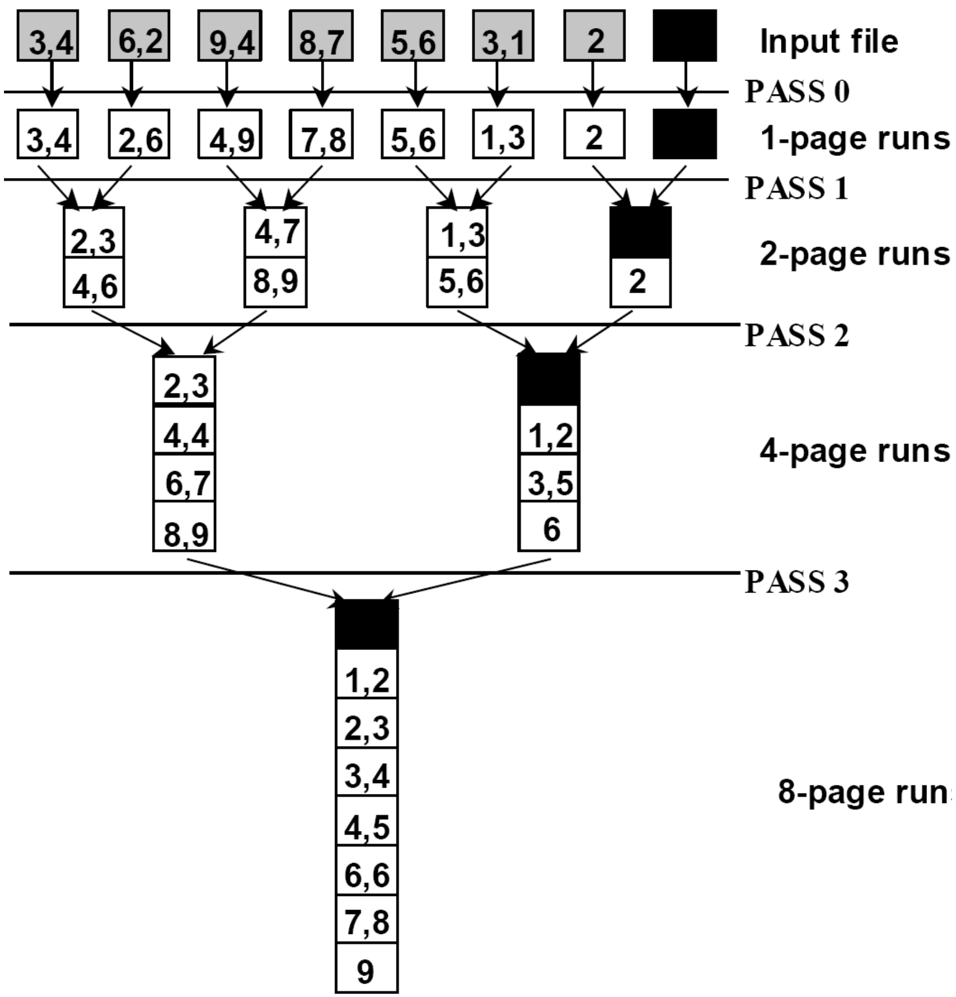
Two-Way Sort

- Requires 3 buffers
  - Pass 1: Read a page, sort it, write it.
    - only one buffer page is used
  - Pass 2, 3, …, etc.:
    - three buffer pages are used



[Ramakrishnan & Gehrke, 2003]

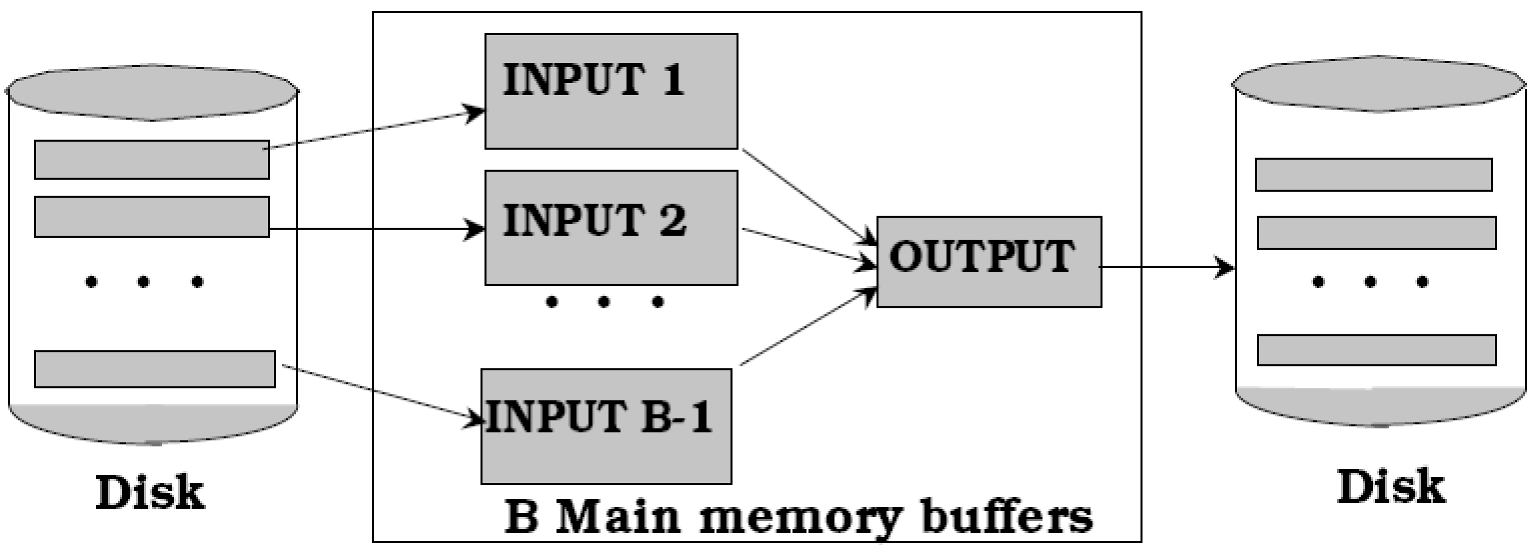
Two-Way Merge Sort



[Ramakrishnan & Gehrke, 2003]

General Merge Sort

- General Merge Sort with more than 3 buffer pages
  - Pass 0: use B buffer pages. Produce sorted runs of $\lceil N/B \rceil$ pages each (N is the number of pages of the input file)
  - Pass 2, …, etc.: merge B-1 runs
- Costs: $\underbrace{2N}_{\text{Cost for 1 pass}} * \underbrace{(1+\lceil \log_{B-1}\lceil N/B \rceil \rceil)}_{\text{Number of passes}}$



[Ramakrishnan & Gehrke, 2003]

Number of Passes for External Merge Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

[Ramakrishnan & Gehrke, 2003]

Other Possibilities for Sorting

- Internal Sorting
  - Quicksort vs. Heapsort
- Using B+ Trees for Sorting
  - If clustered ⇨ Good idea!
  - If not clustered (data records not sorted according index attribute) ⇨ Usually a bad idea!

Running Example

- **Extended Schema:**

OFFICE
<u>floor</u>
<u>room</u>
<u>eno</u>

DEPT
<u>dno</u>
dname
mgr

EMPL
<u>eno</u>
name
marstat
salary

TASK
<u>eno</u>
<u>pno</u>
tname
due_date

PROJECT
<u>pno</u>
pname

- **Sizes:**
  - TASK: each tuple 40 bytes, 100 tuples/page (p_T), 1000 pages (M)
  - EMPL: each tuple 50 bytes, 80 tuples/page (p_E), 500 pages (N)

- **Costs:**
  - I/O costs for fetching 1 page
  - O-Notation for complexity of operations, M denotes number of pages
  - No other costs are considered (e.g., for processing of data or data output)

2.2.2 Selection

- Query: $\sigma_{\text{tname=„UI Design“}}(\text{TASK})$

- No index, unsorted data:
  - Scan the entire relation, costs O(M)

- No index, sorted data (according to selection attribute)
  - Binary search, costs $O(\log_2 M)$
  - If range is selected, e.g. $\sigma_{\text{due\_date}>01.09.2014}(\text{TASK})$

Costs for retrieving tuples have to be added!

Index-based Selection

- Selection using an Index:
  - Costs: depend on # of qualifying tuples and clustering
  - Costs finding qualifying data entries (typically small) +
    costs of retrieving records (could be large without clustering)
  - Assumption in example:
    - uniform distribution of tname
    - about 10% of tuples qualify (100 pages, 10000 tuples)

  → With clustered index: little more than 100 I/Os
  → Unclustered: up to 10000 I/Os!

- Important refinement for unclustered indexes :

  1. Find qualifying data entries.
  2. Sort the IDs of the data records to be retrieved.
  3. Fetch IDs in order → ensures that each data page is looked at just once
     (though the # of such pages is likely to be higher than with clustering, in worst case one page for each qualifying tuple).

General Selection Conditions

Attr **op** Const   or   Attr1 **op** Attr2

(and their boolean combinations, **op** in {<,>,<=,>=,=,<>})

<u>First approach:</u>

1. Find most selective access path (index or file scan, requiring the fewest page I/Os, reduces # of retrieved tuples)
2. Retrieve tuples using it
3. Apply any remaining terms that don't match the index (discards some retrieved tuples, but does not affect # of tuples/pages fetched)

Example:

- Select condition: due_date<01.09.2014 AND pno=3 AND eno=5
- Option 1: Use B+ tree index on due_date
  → eno=5 and pno=3 must be checked for each retrieved tuple
- Option 2: use hash index on <pno, eno>
  → due_date<01.09.2014 must then be checked.

General Selection Conditions

Second Approach:

If we have two or more matching indexes:

1. Get sets of IDs of data records using each matching index.
2. Intersect these sets of IDs (we'll discuss intersection soon!)
3. Retrieve the records and apply any remaining terms.

Example:

- Select condition: due_date<01.09.2014 AND pno=3 AND eno=5
- Use B+ tree index on due_date and index on eno
  1. retrieve IDs of records satisfying due_date<01.09.2014 using index 1
  2. retrieve IDs of records satisfying eno=5 using index 2
  3. intersect
  4. retrieve records
  5. check pno=3

2.2.3 Projection

- Query: $\Pi_{eno,pno}(TASK)$

- Effect
  1. Remove unwanted attributes
  2. Eliminate duplicate tuples

- Projection based on sorting
  1. Remove unwanted attributes and store in temporary relation
  2. Sort temporary relation
  3. Scan result, comparing adjacent tuples, and discard duplicates

  → Costs: O(M log M)

- Improvement: modify external merge sort
  - Pass 0: as before **<u>and</u>** eliminate unwanted attributes
  - Pass 1,…,n: Merge previous runs and eliminate duplicates

Projection with Hashing or Indexes

- **Projection based on hashing**
  - Partitioning phase:
    - Read R using one input buffer
    - For each tuple, discard unwanted fields, apply hash function h1 to choose one of B-1 output buffers
    - Result is B-1 partitions (of tuples with no unwanted fields) → two tuples from different partitions guaranteed to be distinct
  - Duplicate elimination phase:
    - For each partition, read it and build an in-memory hash table, using hash function h2 (<> h1) on all fields, while discarding duplicates.
    - If partition does not fit in memory, apply hash-based projection algorithm recursively to this partition
- **Projection using indexes**
  - If index contains all retained attributes, index can be accessed
  - Much smaller set of pages

2.2.4 Join

- Query: $T \bowtie_{eno=eno} E$

Example: Names of single employees who work on design tasks and make less than 40,000

SQL:

```
SELECT DISTINCT e.name FROM EMPL e, TASK t
WHERE  e.salary < 40,000 AND e.marstat = 'single'
AND    t.tname= 'design' AND e.eno = t.eno
```

Relational Algebra (RA):

$$\pi_{name}(\sigma_{salary<40{,}000 \wedge marstat='single'}(EMPL) \bowtie \sigma_{tname='design'}(TASK))$$

Nested Loop Join

- Iterate over both relations using two nested loops
- Check join and selection conditions in inner loop
- If ok, add projected attribute(s) to result set
- For each tuple in outer relation T, entire inner relation E is scanned
- **Costs:** $M + p_T * M * N = 1000 + 100*1000*500 =$ <u>50.001.000 I/Os</u>

```
ANSWER:=[];
FOR EACH t in TASK DO
  FOR EACH e IN EMPL DO
    IF  e.salary<40,000 AND e.marstat='single' AND
        t.tname=‘design' AND t.eno=e.eno
    THEN ANSWER:+[<e.name>];
```

Improved Nested Loop Join

Heuristics to improve query execution: Selection before join!

1. Scan one relation, check selection conditions, put result into temporary buffer
2. Scan second relation, check join & selection condition using intermediate result from temporary buffer, create result set

```
ENOLIST:=[];
FOR EACH t IN TASK DO
  IF t.tname=‘design' THEN ENOLIST:+[<t.eno>];
ANSWER:=[];
FOR EACH e in EMPL DO
IF  e.salary<40,000 AND e.marstat='single'
    THEN FOR EACH t IN ENOLIST DO
      IF t.eno=e.eno THEN ANSWER:+[<e.name>];
```

Costs: if the result of $\sigma_{tname='design'}(\pi_{eno}(T))$ fits into buffer → M + N

Index Nested Loop Join

```
ANSWER:=[];
FOR EACH t IN TASK DO
    Lookup t.eno in index on Empl.eno, get tuple e from EMPL
        IF found THEN ANSWER:+[<t,e>];
```

- If there is an index on the join column of one relation (say E), we can make it the inner and exploit the index
- **Costs:** M + (($M*p_T$) * cost of finding matching E tuples)
- Costs of probing E index for each T tuple:
  - hash index: about 1.2
  - B+ tree: 2-4
  - Cost of then finding E tuples depends on clustering.
- Clustered index: 1 I/O (typical) for each T tuple
- Unclustered: up to 1 I/O per matching E tuple

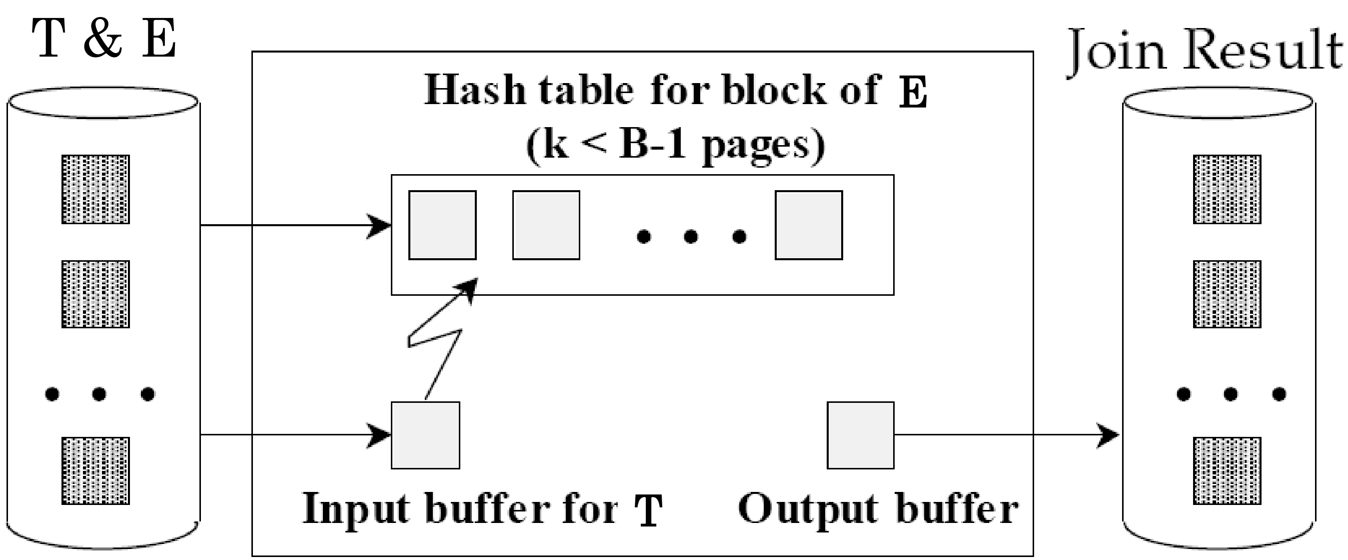
Page-oriented Nested Loop Join

- For each page of T, get each page of E
- Write out matching pairs of tuples <t, e>, (where t is in T-page and e is in E-page)
- **Costs:**
  - M + M*N = 1000 + 1000*500 = <u>501.000 I/Os</u>
  - If smaller relation (E) is outer, costs = 500 + 500*1000

    = <u>500.500 I/Os</u>

Block Nested Loop Join

- Use 1 page as input buffer to scan inner T and 1 page as output buffer
- Use all remaining pages to hold “block” of outer E.
- For each matching tuple e in E-block, t in T-page,
  add <e, t> to result
- Then read next E-block, scan T, etc.
- **Costs:** Scan of outer + #outer blocks * scan of inner
  $M + \lceil M/(B-2) \rceil * N$



Sort-Merge Join (1)

1. Sort T and E on the join column

2. Scan T and E to do a “merge” (on join column)
   - Advance scan of T until current T-tuple >= current E tuple
   - Advance scan of E until current E-tuple >= current T tuple
   - Do this until current T tuple = current E tuple.
     → At this point, all T tuples with same value in T_i (current T group) and all E tuples with same value in E_j (current E group) match
     → Output <t, e> for all pairs of such tuples.
     → Then resume scanning T and E

3. Output result tuples

EMPL

eno	name	...
2	Bob	
3	Zoe	
6	Tom	
4	Joe	

TASK

eno	pno	...
4	1	
6	5	
4	3	
3	2	

EMPL

eno	name	...
2	Bob	
3	Zoe	
4	Joe	
6	Tom	

TASK

eno	pno	...
3	2	
4	1	
4	3	
6	5	

EMPL_TASK

eno	name	pno	...
3	Zoe	2	
4	Joe	1	
4	Joe	3	
6	Tom	5	

Sort-Merge Join (2)

- **Costs:**
  - T is scanned once
  - Each E group is scanned once per matching T tuple. (Multiple scans of an E group are likely to find needed pages in buffer)

  → M log M + N log N + (M+N)

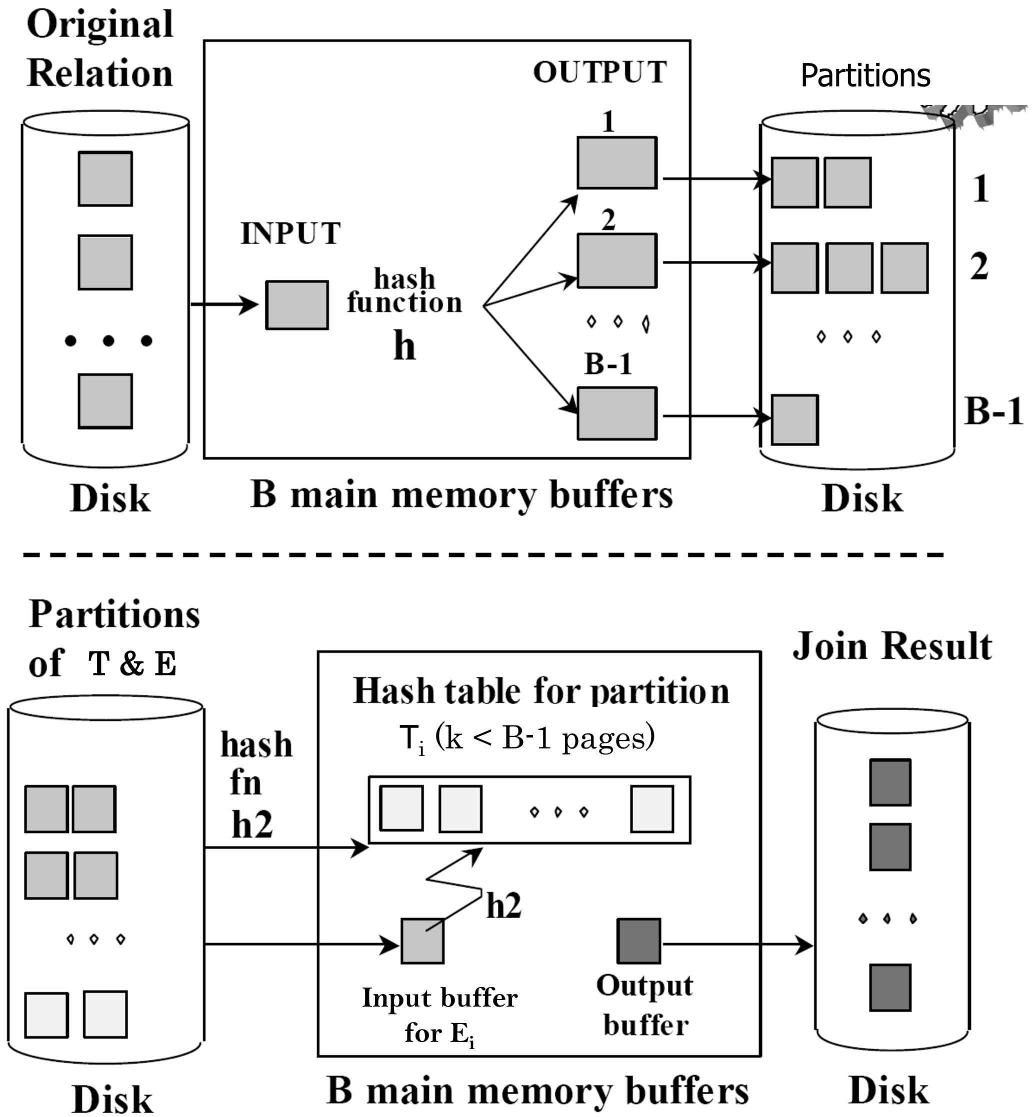
- We can combine the merging phases in the *sorting of* T and E with the merging required for the join.

- In practice, costs of sort-merge join, like the costs of external sorting, is *linear.*

Hash Join

- Partition both relations using hash fct h. T tuples in partition i will only match E tuples in partition i.
- Read 1 partition of T, hash it using h2 (<> h). Scan matching partition of E, search for matches.
- **Costs:** In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.

→ 3(M+N)



General Join Conditions

- Equalities over several attributes (e.g., T.eno=E.eno AND T.tname=E.name):
  - **Index Nested Loop:** build index on <eno, name> (if E is inner); or use existing indexes on eno or name.
  - **Sort-Merge and Hash Join:** sort/partition on combination of the two join columns.

- Inequality conditions (e.g., T.tname < E.name):
  - For **Index Nested Loop**, need (clustered!) B+ tree index.
    - Range probes on inner; #matches likely to be much higher than for equality joins.
  - **Hash Join, Sort Merge Join** not applicable.
  - **Block Nested Loop Join** quite likely to be the best join method here.

Comparison of Join Implementations (1)

Query: TASK $\bowtie_{eno=eno}$ EMPL

Name	**Description**	**Costs**	**Example**	**Time** **(10ms per I/O)**
Simple Nested Loop	Inner Relation is scanned for each tuple in outer relation	$M + p_T * M * N$	1000+ 100*1000*500 ≈ $5*10^7$ I/Os	140 h
Page-Oriented Nested Loop	Inner Relation is scanned only for each page of outer relation	M+M*N	1000 + 1000*500 ≈ $5*10^5$ I/Os	1.4 h
Block Nested Loop	Inner Relation is scanned for each block of pages of outer relation	$M + \lceil M/(B-2) \rceil * N$	1000 +10*500 = 6000 I/Os (assuming B=102, assuming B=336 → 2500 I/Os)	≈ 1 min

Comparison of Join Implementations (2)

Name	Description	Costs	Example
Index Nested Loop	Nested Loop algorithm with index on a relation	$M + p_R * M *$ (costs for index and data access)	1000 + 100 * 1000 (1.2 + 1) = 221.000 I/Os (assuming hash index on eno of EMPL with 1.2 I/Os on average and 1 I/O for data access; page has to be fetched, not buffered)
			500 + 80 * 500 * 1.2 = 48.500 I/Os (assuming hash index on eno of TASK, without retrieving data from TASK) **Costs for retrieving data from TASK:** >1 matching tuple in TASK for each employee (assume 2.5 tuples on avg.) **Clustered:** 1 I/O per tuple in EMPL = 40,000 I/Os (all matching tuples in TASK on 1 page) **Unclustered:** 2.5 I/Os per tuple in EMPL = 100,000 I/Os **Total Costs:** 88.500 I/Os to 148.500 I/Os

Comparison of Join Implementations (3)

Name	Description	Costs	Example
Sort-Merge Join	First sort relations on join column, then scan to merge them. 2-Phase-Sort: in each phase, read and write each page	O(M log M) + O(N log N) Sort + M + N Merge	2*2*1000 + 2*2*500 + 1000 + 500 = 7500 I/Os
Hash Join	Partition both relations using a hash function; hash partitions and find matches in corresponding partitions	2 (M + N) + Partitioning M + N Matching = 3 (M + N)	= 3 * (1000 + 500) = 4.500 I/Os

2.2.5 Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar; we'll do union (compare with projection)
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - Alternative: Merge runs from Pass 0 for both relations.
- Hash based approach to union:
  - Partition T and E using hash function h.
  - For each E-partition, build in-memory hash table (using h2),
  - scan corr. T-partition and add tuples to table while discarding duplicates.

2.2.6 Aggregate Operations (1)

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

Example: Average salary of all single employees.

```
SELECT AVG(e.salary) AS avgsinglesalary
FROM EMPL e
WHERE e.marstat="single"
```

Aggregate Operations (2)

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given a tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, we can do an index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Example: Average salary of employees per department.

```
SELECT AVG(e.salary) AS avgdepsalary,
FROM EMPL e, DEPT d
WHERE e.dno = d.dno
GROUP BY d.dname
```

Summary of 2.2

- Order of data on disk or in intermediate results is important
  - ➔ Clustering
  - ➔ Sorting

- Block-oriented access in operators improves performance (e.g., block nested loop join or merge-sort)

- Buffer pool affects query evaluation
  - Buffer size
  - Replacement policy
  - Buffer has to be shared if operations are executed in parallel

Review Questions

- What are the four query languages which we discussed for relational database systems?
- What is a relationally complete language?
- What is the difference between a procedural and a declarative query language?
- Why is sorting an important operation in database systems?
- How do you sort a huge amount of data (in external memory) efficiently?
- What is the complexity of the projection operation?
- How can a join be implemented (efficiently)?
- What is the improvement in block nested loop join compared to the „normal" nested loop join?
- Explain the hash join/sort-merge join! Complexity?