Database Development and Design (CPT201)

Lecture 4a: Relational Algebra

Dr. Wei Wang
Department of Computing

Learning Outcomes

- Six basic operators
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: -
  - Cartesian product: x
  - rename: ρ
- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment

Relational Algebra, What and Why?

- Similar to normal algebra (as in 2+3*x-y), except we use relations as values instead of numbers, and the operations and operators are different.
- Not used as a query language in actual DBMSs. (SQL instead.)
- The inner, lower-level operations of a relational DBMS are, or are similar to, relational algebra operations. We need to know about relational algebra to understand query execution and optimisation in a relational DBMS.
- Some advanced SQL queries requires explicit relational algebra operations, most commonly *outer join*.
- Relations are seen as *sets of tuples*, which means that no duplicates are allowed. SQL behaves differently in some cases. Remember the SQL keyword distinct.
- SQL is *declarative*, which means that you tell the DBMS *what* you want, but not *how* it is to be calculated. A C++ or Java program is *procedural*, which means that you have to state, step by step, exactly how the result should be calculated. Relational algebra is (more) procedural than SQL. (Actually, relational algebra is mathematical expressions.)

Concepts and operations from set theory

- Relations in relational algebra are seen as sets of tuples, so we can use basic set operations.
  - set
  - element
  - no duplicate elements (but: multiset = bag)
  - no order among the elements (but: ordered set)
  - subset
  - proper subset (with fewer elements)
  - superset
  - union
  - intersection
  - set difference
  - Cartesian product (cross-product)

Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation
- Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p(E_1)$, P is a predicate on attributes in E_1
  - $\prod_S(E_1)$, S is a list consisting of some of the attributes in E_1
  - $\rho_x(E_1)$, x is the new name for the result of E_1

Select Operation– Example

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where p is a formula in propositional calculus consisting of **terms** connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
Each **term** is one of:

<attribute> *op* <attribute> or <constant>

where *op* is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection:

$$\sigma_{branch\_name="Perryridge"}(account)$$

Project Operation – Example

Relation *r*

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

$\prod_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation:

$$\Pi_{A_1, A_2, \ldots, A_k}(r)$$

where A_1, A_2 are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: to eliminate the *branch_name* attribute of *account* (branch_name, account_name, balance)

$$\Pi_{account\_number,\ balance}(account)$$

Union Operation – Example

Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

Union Operation

- Notation: $r \cup s$
- Defined as:

  $$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For $r \cup s$ to be valid.

  1. r, s must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)

- Example: to find all customers with either an account or a loan

  $$\Pi_{customer\_name}(depositor) \cup \Pi_{customer\_name}(borrower)$$

Set Difference Operation – Example

Relations r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r - s$

A	B
α	1
β	1

Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- Set differences must be taken between **compatible** relations.
  - r and s must have the same arity
  - attribute domains of r and s must be compatible

Cartesian-Product Operation – Example

Relations *r*, *s*

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r x *s*

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

- Notation $r \times s$
- Defined as:

  $r \times s = \{t\, q \mid t \in r \textbf{ and } q \in s\}$
- Assume that attributes of r(R) and s(S) are disjoint, that is, $R \cap S = \varnothing$.
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used.

Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$

$r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$\sigma_{A=C}(r \times s)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\rho_x(E)$$

returns the expression E under the name X

- If a relational-algebra expression E has arity n, then

$$\rho_{x(A_1,A_2,\ldots,A_n)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to A_1, A_2, …., A_n.

Banking Example

branch (branch_name, branch_city, assets)

customer (customer_name, customer_street, customer_city)

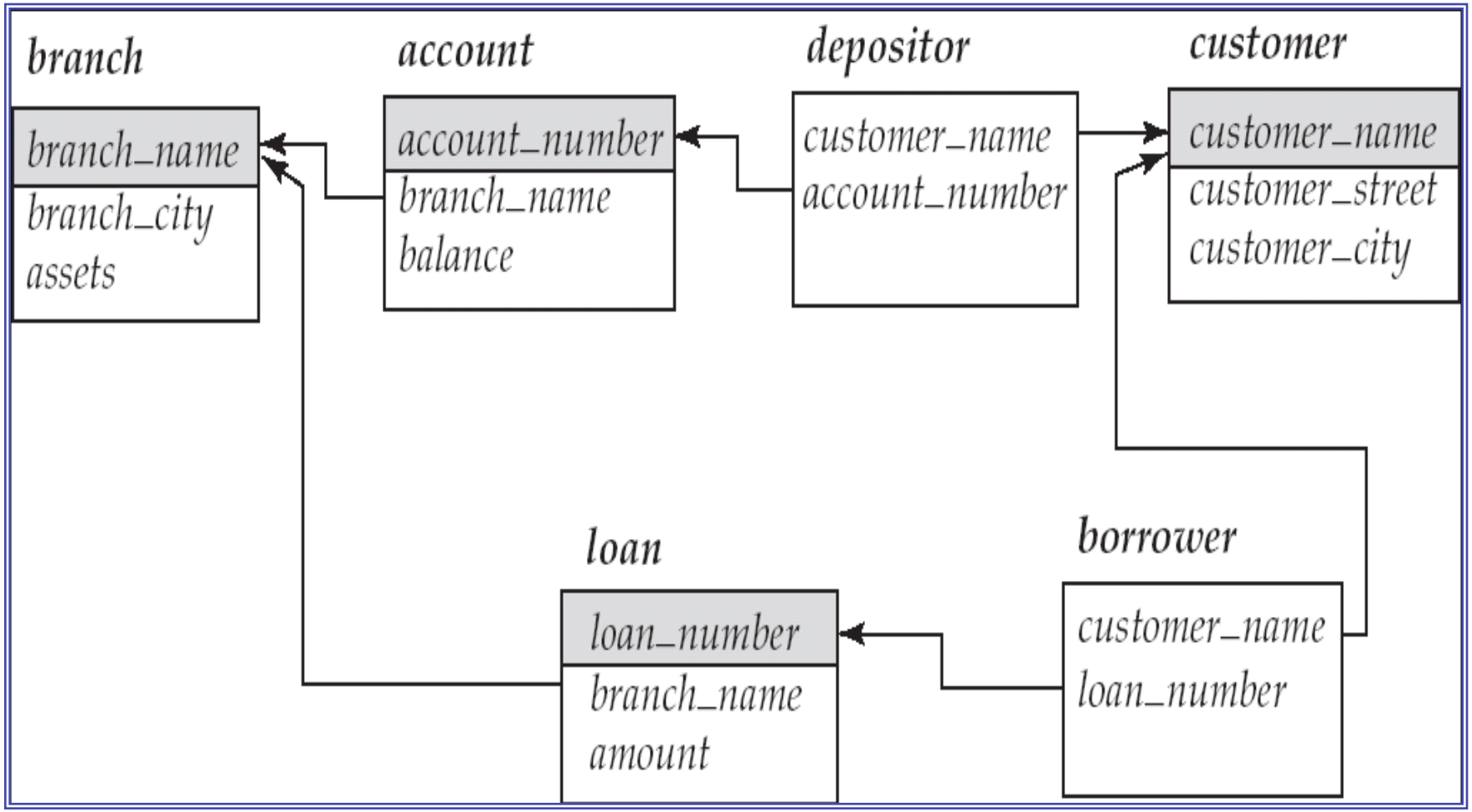
account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

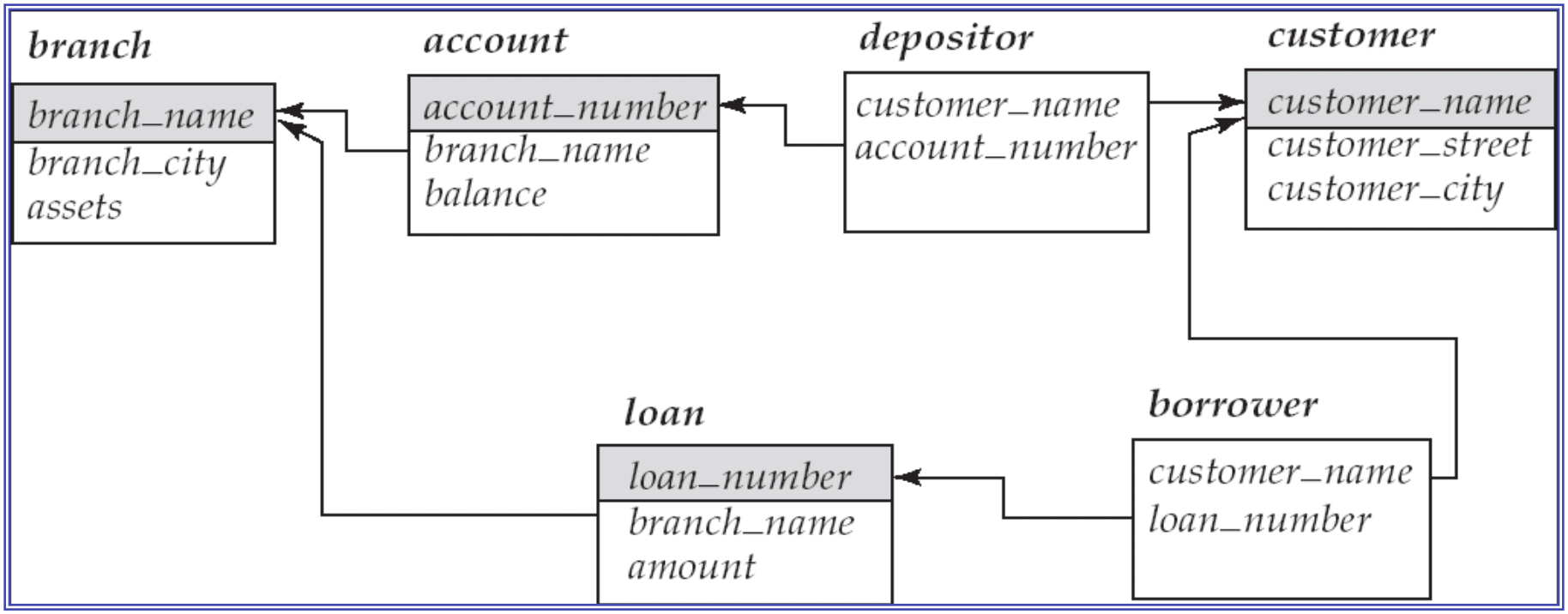
depositor (customer_name, account_number)

borrower (customer_name, loan_number)

Banking Example



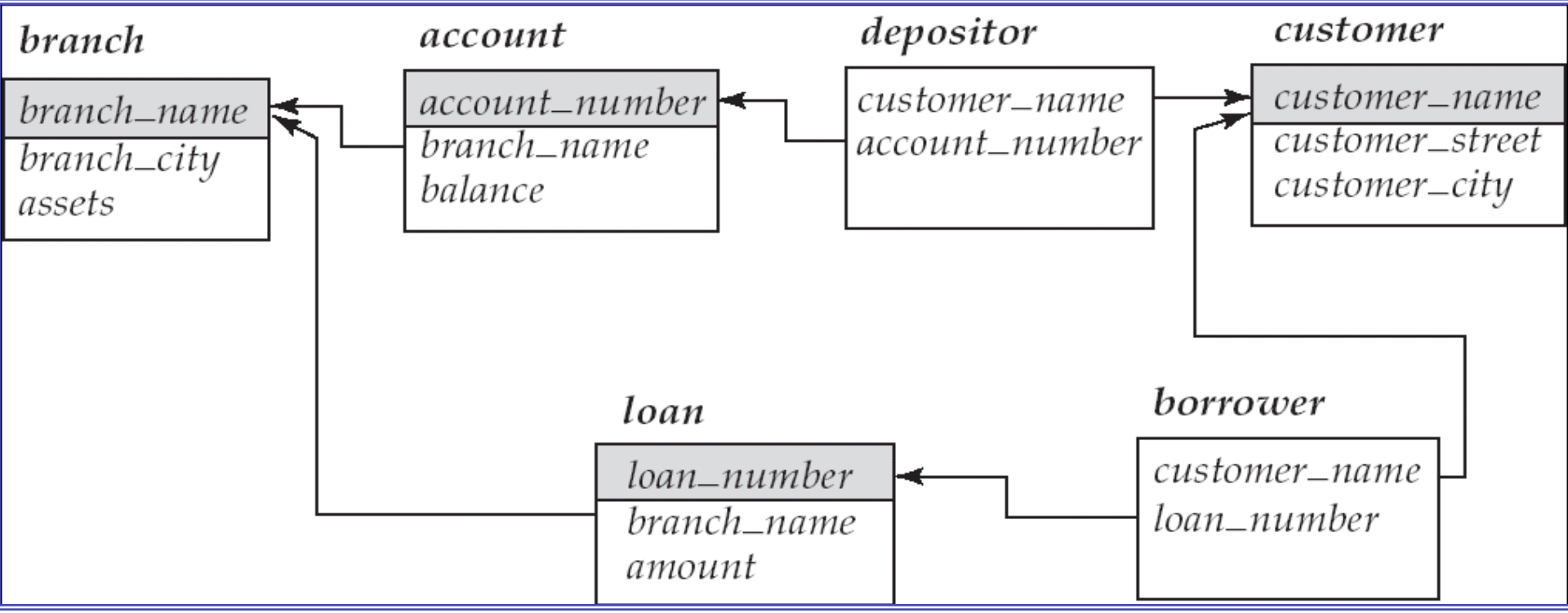
Example Queries



Find all loans of over $1,200

$\sigma_{amount > 1,200}(loan)$

Example Queries cont'd



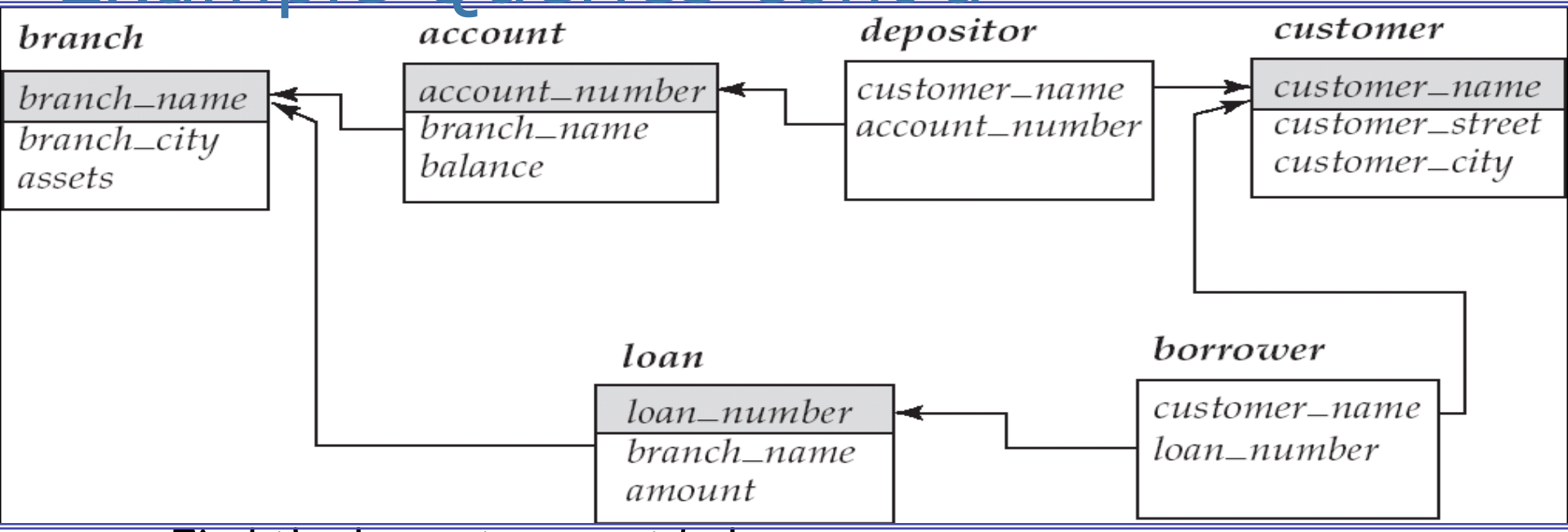
Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{\text{customer\_name}} (\sigma_{\text{branch\_name} = \text{"Perryridge"}} (\sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}} (\text{borrower} \times \text{loan})))$$

OR

$$\Pi_{\text{customer\_name}} (\sigma_{\text{loan.loan\_number} = \text{borrower.loan\_number}} ((\sigma_{\text{branch\_name} = \text{"Perryridge"}} (\text{loan})) \times \text{borrower}))$$

Example Queries cont'd



- Find the largest account balance
  - Strategy:
    - Find those balances that are *not* the largest
      - Rename *account* relation as *d* so that we can compare each account balance with all others
    - Use set difference to find those account balances that were *not* found in the earlier step.

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_{d}(account)))$$

Additional Operations

- We define additional operations that do not add any power to the relational algebra, but that simplify common queries.
  - Set intersection
  - Natural join
  - Division
  - Assignment

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$
- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

Set-Intersection Operation – Example

Relation *r*, *s*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cap s$

A	B
α	2

Natural-Join Operation

- Notation: $r \bowtie s$
- Let r and s be relations on schemas R and S respectively. Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s
- Example:

  $R = (A, B, C, D)$

  $S = (E, B, D)$
  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:

$$\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E} (\sigma_{r.B = s.B \,\wedge\, r.D = s.D} (r \times s))$$

Natural Join Operation – Example

Relations r, s

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

s

r ⋈ s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Division Operation

- Notation: $r \div s$
- Often suited to queries that include the phrase "for all".
- Let r and s be relations on schemas R and S respectively where
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  The result of $r \div s$ is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

  $$r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$$

  Where tu means the concatenation of tuples t and u to produce a single tuple

Division Operation – Example

Relations *r, s*

r

A	*B*
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

s

B
1
2

$r \div s$:

A
α
β

Another Division Example

Relations *r, s*

r

A	*B*	*C*	*D*	*E*
α	a	α	a	*1*
α	a	γ	a	*1*
α	a	γ	b	*1*
β	a	γ	a	*1*
β	a	γ	b	*3*
γ	a	γ	a	*1*
γ	a	γ	b	*1*
γ	a	β	b	*1*

s

D	*E*
a	*1*
b	*1*

$r \div s$

A	*B*	*C*
α	a	γ
γ	a	γ

Division Operation cont'd

- Property
  - Let $q = r \div s$
  - Then q is the largest relation satisfying $q \times s \subseteq r$
- Definition in terms of the basic algebra operation

  Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

$$r \div s = \prod_{R-S}(r) - \prod_{R-S}((\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r))$$

  To see why
  - $\prod_{R-S,S}(r)$ simply reorders attributes of r
  - $\prod_{R-S}((\prod_{R-S}(r) \times s) - \prod_{R-S,S}(r))$ gives those tuples t in $\prod_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.

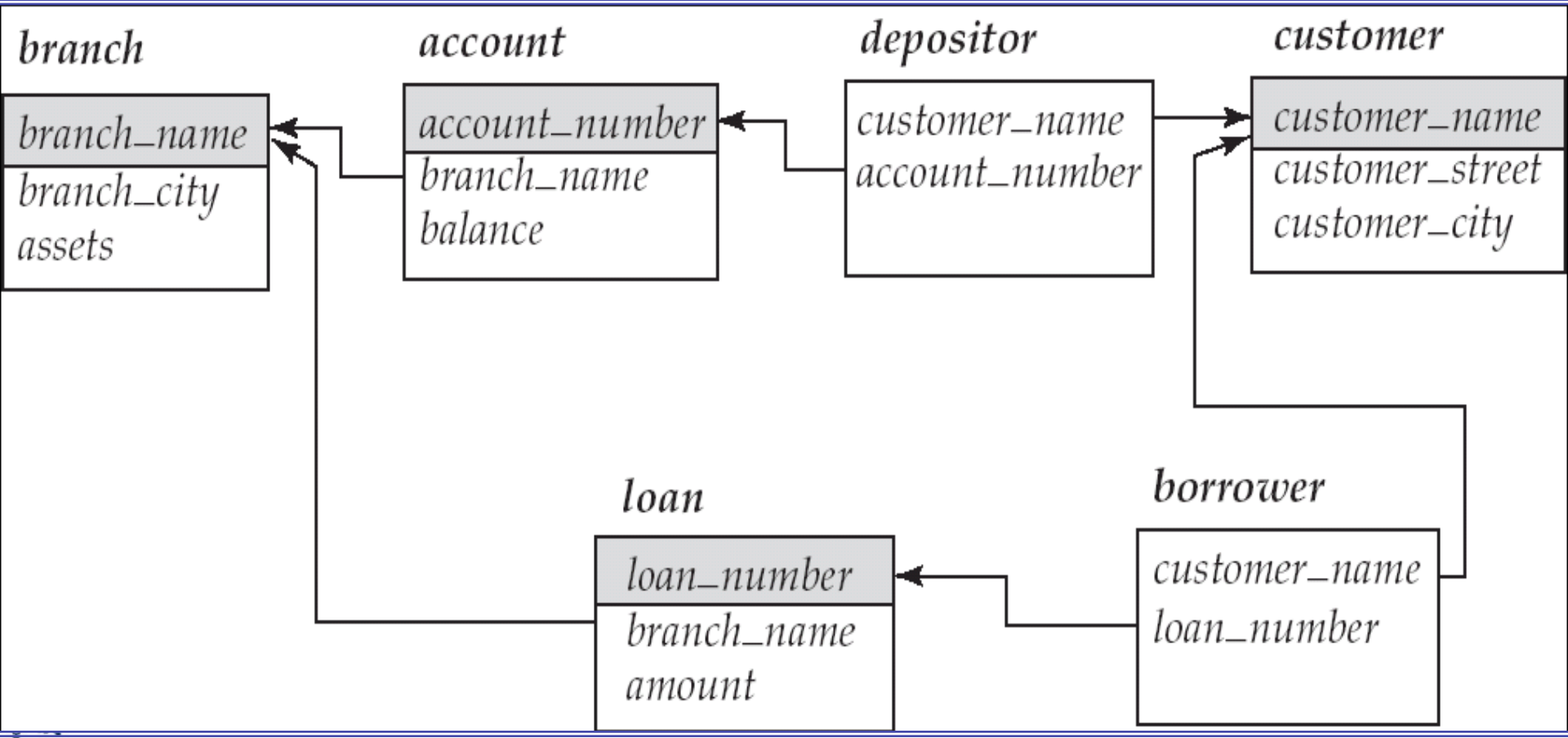
Assignment Operation

- The assignment operation (←) provides a convenient way to express complex queries.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

$$temp1 \leftarrow \prod_{R-S} (r)$$
$$temp2 \leftarrow \prod_{R-S} ((temp1 \times s) - \prod_{R-S,S} (r))$$
$$result = temp1 - temp2$$

  - The result to the right of the ← is assigned to the relation variable on the left of the ←.
  - May use variable in subsequent expressions.

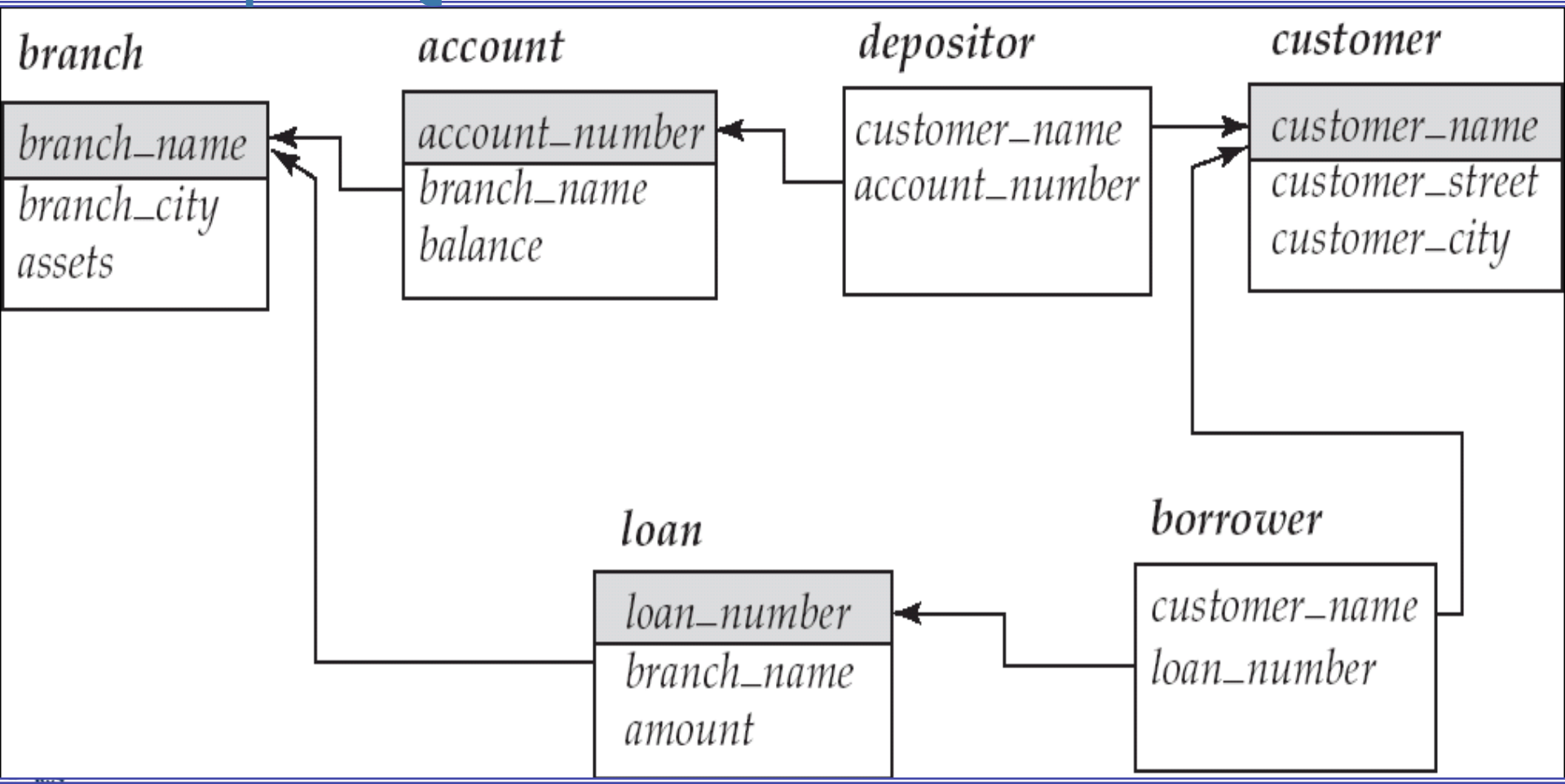
Example Queries cont'd



Find the names of all customers who have both a loan and an account at bank.

$$\Pi_{customer\_name}(borrower) \cap \Pi_{customer\_name}(depositor)$$

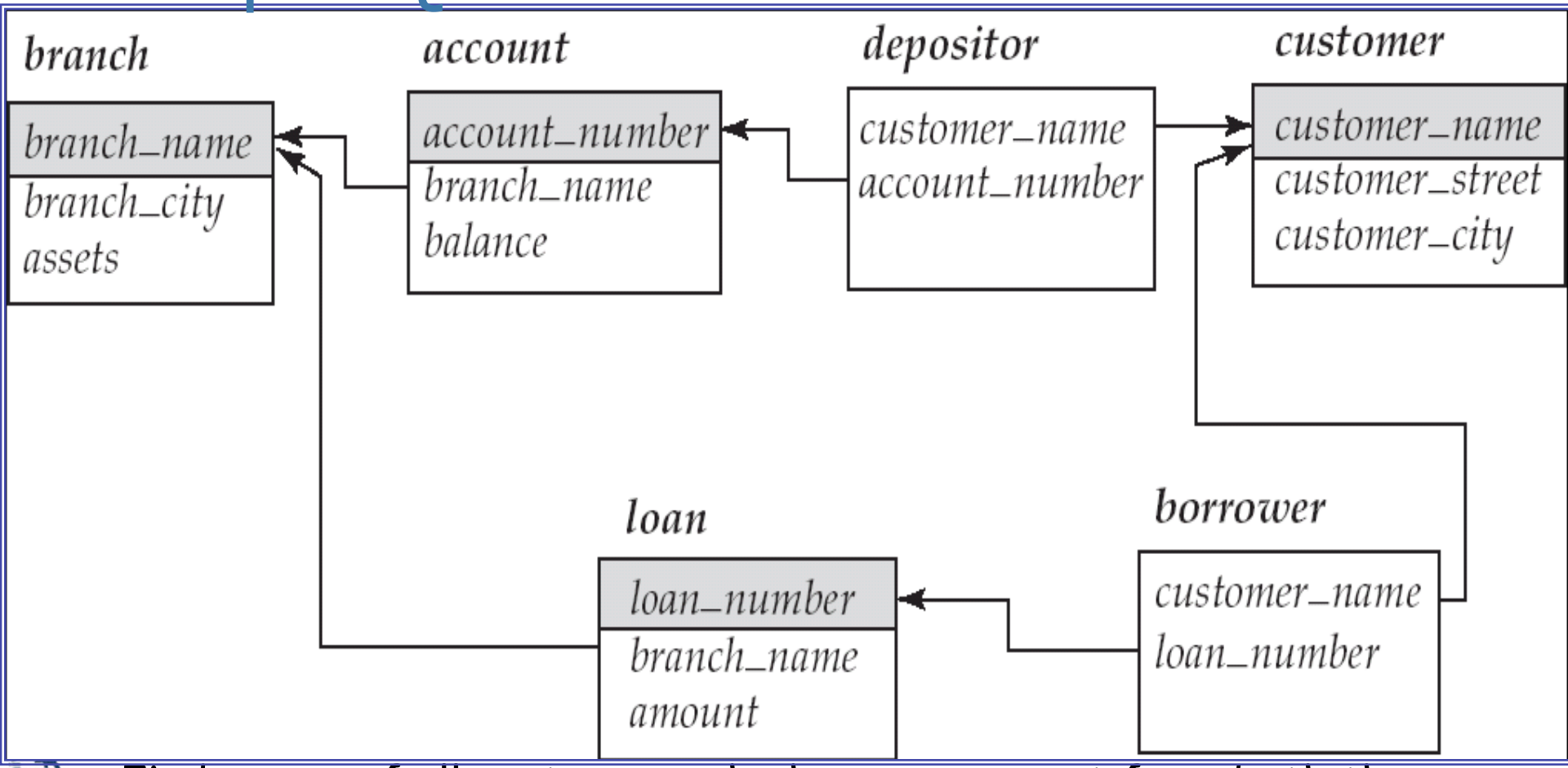
Example Queries cont'd



Find the name of all customers names, their load numbers and loan amount

$$\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan)$$

Example Queries cont'd

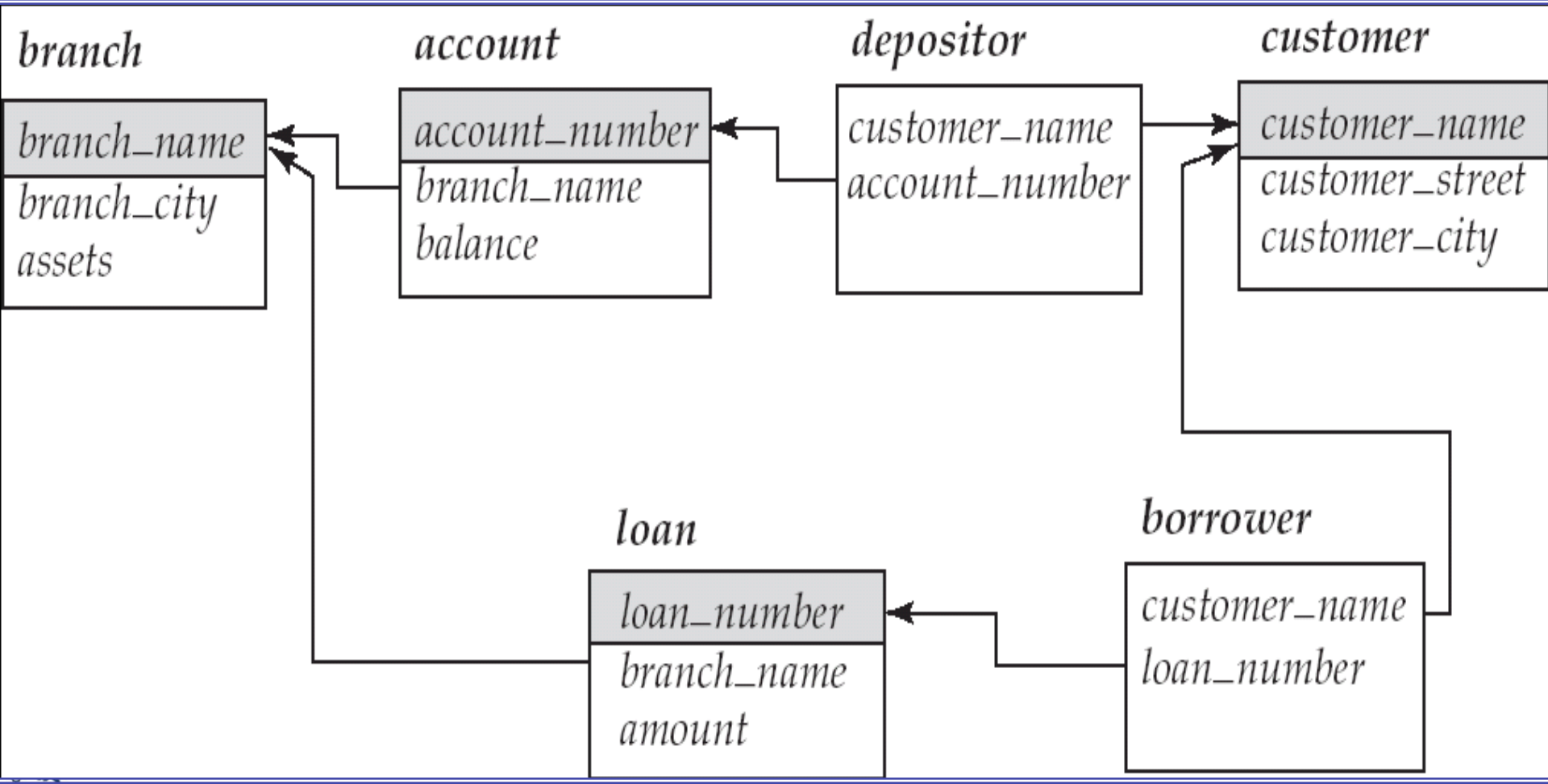


Find names of all customers who have an account from both the "Downtown" and the "Uptown" branches.

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Downtown"}}(depositor \bowtie account)) \cap$$

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Uptown"}}(depositor \bowtie account))$$

Example Queries cont'd



Find names of all customers who have an account at all branches located in Brooklyn city.

$$\Pi_{customer\_name,\ branch\_name}(depositor \bowtie account) \div \Pi_{branch\_name}(\sigma_{branch\_city = \text{"Brooklyn"}}(branch))$$

End of Lecture

- Summary
  - Basic relational algebra operators
  - Additional relational algebra operators
  - Example queries

- Reading
  - Textbook 6th edition, chapter 2.6, 6.1
  - Textbook 7th edition, chapter 2.6

Database Development and Design (CPT201)

Lecture 4b:
Query Evaluation - Selction

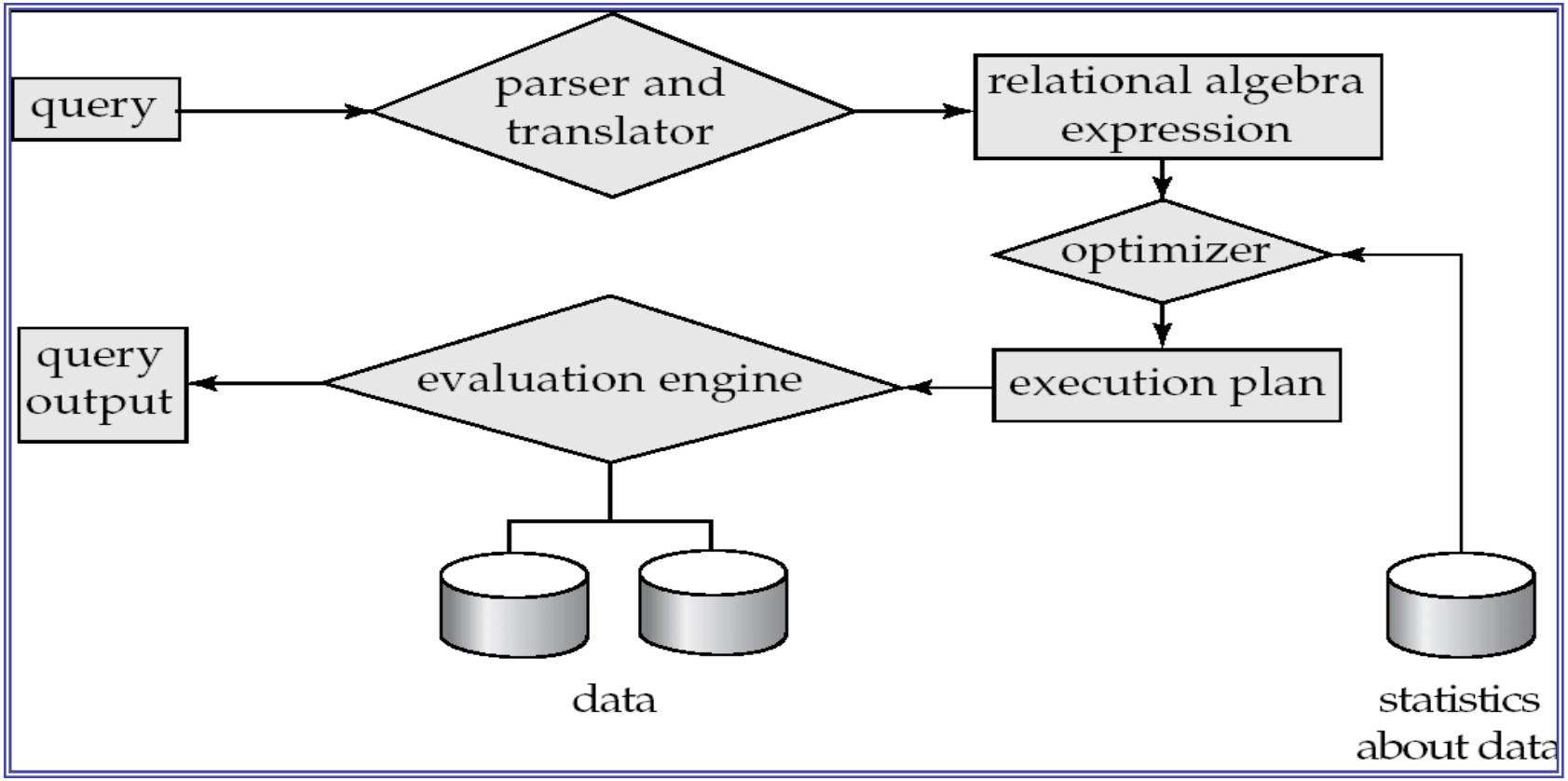
Dr. Wei Wang

Department of Computing

Learning Outcomes

- Basic steps in query processing
- How to measure query costs
- Algorithms for evaluating relational algebra operations (Selection and Projection)
- External Merge Sort

Basic Steps in Query Processing



Parsing and translation

- Translate the query into its internal form.
- This is then translated into relational algebra.
  - (Extended) relational algebra is more compact, and differentiates clearly among the various different operations
- Parser checks syntax, verifies relations
- This is a subject for *compilers*

Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
  - The bulk of the problem lies in how to come up with good evaluation plans!

Optimisation

- A relational algebra expression may have many equivalent expressions, e.g.,
  - $\sigma_{balance<2500}(\prod_{balance}(account))$
  - $\prod_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using 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**, e.g.,
  - Plan 1: use an index on *balance* to find accounts with balance < 2500,
  - Plan 2: perform linear scan and discard accounts with balance ≥ 2500
- **Query Optimisation**: Amongst all equivalent evaluation plans choose the one with the lowest cost.
  - Cost is estimated using statistical information from the database catalog, e.g. number of tuples in each relation, size of tuples, number of blocks in a relation, etc.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering a query
  - Many factors contribute to time cost, such as *disk accesses, CPU,* or even network *communication*
- For simplicity, we just use *number of block transfers* from/to disk and *number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for *b* block transfers plus *s* seeks: $b * t_T + s * t_S$
- We do not include cost to writing final output to disk in the cost formulae (with some exceptions, will see later)
- We ignore CPU costs for simplicity, as they tend to be much lower
- Evaluating the cost of an algorithm in terms of block transfers and seeks is substantially different from that in term of number of steps.

Access Paths

- Possible access paths for selections:
  - Scan the file records.
  - Scan the index
- Selectivity of an access path is the number of block I/Os needed to retrieve the tuples satisfying the desired condition.
  - Obviously, we want to use the most selective access path (with the fewest page/block I/Os).
- Access path using index may not be the most selective!
  - Why not? ($\sigma_{bid \neq 10}$ Reserves, non-clustering index on bid).

Selection Operation

- Notation: $\sigma_p(r)$
- p is the selection predicate
- Defined by:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

in which p is a formula of propositional calculus of terms connected by: ∧ (**and**), ∨ (**or**), ¬ (**not**)
Each term is of the form:

<attribute> *op* [<attribute> or <constant>]

where *op* can be one of: =, ≠, >, ≥. <. ≤

- Selection example:

$\sigma_{branch\text{-}name='Perryridge'}(account)$

Evaluation of Selection Operation

- **File scan** – search algorithms that scan files and retrieve records that fulfill a selection condition.
  - blocks of a relation are stored contiguously
- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.

Algorithm for Selection Operation (File scan: *linear search*)

- Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation *r*
  - If selection is on a key attribute, can stop on finding record (Linear Search, Equality on Key)
    - Average cost = (b_r/2) block transfers + 1 seek
  - This linear search can be always applied, regardless of:
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Algorithms for Selection Operation (File scan: *binary search*)

- Applicable only if the selection is an equality comparison on the attribute on which file is ordered.
  - Cost estimate (number of block transfers and seeks):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - Add transfer cost of the number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost later
  - If b_r is not too big, then most likely binary search doesn't pay.
    - Note that t_T is several (say, 40) times smaller than t_S
    - Which of the two algorithms needs to be wisely chosen for a specific query at hands.

Selections Using Indices (*primary index on candidate key, equality*).

- Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$

  where h_i denotes the height of the index
- Recall that the height of a B+-tree index is at most $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$, where n is the number of pointers per node and K is the number of search keys.
  - E.g. for a relation r with 1,000,000 different search key, and with 100 index entries per node, $h_i = 4$
  - Unless the relation is really small, this algorithms always "pays" when indexes are available.

Selections Using Indices (*primary index on nonkey, equality)*

- Retrieve possibly multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices (*equality on key and non-key of secondary index)*

- Retrieve a single record if the search-key is a candidate key
  - $Cost = (h_i + 1) * (t_T + t_S)$
- Retrieve multiple records if search-key is not a candidate key
  - each of *n* matching records may be on a different block
  - Cost at most is: $(h_i + n) * (t_T + t_S)$
  - Can be very expensive if n is big! Note that it multiplies the time for seeks by *n*.

Comparative Selections $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$

- Using a linear file scan or binary search just as before
- Using primary index, comparison
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple ≥ v and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > v;
    - Using the index would be useless, and would requires extra seeks on the index file.
- Using secondary index, comparison
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry ≥ v and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > v
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record (a lot!)
    - Linear file scan may be much cheaper!!!!

Conjunctive Selections

$\sigma_{\theta 1 \wedge \theta 2 \wedge \cdots \wedge \theta n}(r)$

- *using one index*
  - Select θ_i and previous algorithms that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
  - In this case the choice of the first condition is crucial!
    - One must use estimates to know which one is the best.
- *using multiple-key index*
  - Use appropriate composite (multiple-key) index if available.

Disjunctive Selections

$\sigma_{\theta 1 \vee \theta 2 \vee \cdots \vee \theta n}(r)$

- Union of identifiers
  - Use linear scan.
  - Or index scan if *some* conditions have available indices.
    - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
    - Then fetch records from file

Selections With Negation: $\sigma_{\neg\theta}(r)$

- Use linear scan on file
- If an index is applicable to θ
  - Find satisfying records using index
  - $\sigma_{\neg\theta}(r)$ is simply the set of tuples in r that are not in $\sigma_{\theta}(r)$

Duplicate elimination

- Duplicate elimination can be implemented via hashing or sorting.
  - On sorting, duplicates will come adjacent to each other, duplicates can be deleted.
    - *Optimisation:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge (details come later)
  - Hashing is similar – duplicates will come into the same bucket.

Evaluating Projection

- Projection drops columns not in the selected attribute list.
- The expensive part is removing duplicates.

Sorting

- Sorting algorithms are important in query processing at least for two reasons:
  - The query itself may require sorting (**order by** clause)
  - Some algorithms for other operations, like join, set operations and aggregation, require that relations are previously sorted and duplicates removed.
- To sort a relation:
  - We may build an index on the relation, and then use the index to read the relation in sorted order.
    - This only sorts the relation logically, not physically
    - May lead to one disk block access for each tuple.
  - For relations that fit in memory sorting algorithms that you've studied before, like quicksort, can be used.

External Sort-Merge

- For relations that don't fit in memory, special algorithms that take into account the measures in terms of block transfers and seeks, are required.
- Let *M* denote memory size: the number of disk blocks whose contents can be buffered in main memory.
- **Create sorted runs**. Let *i* be 0 initially.
  Repeatedly do the following till the end of the relation:
  (a) Read *M* blocks of relation into memory;
  (b) Sort the in-memory blocks;
  (c) Write sorted data to run file R_i;
  (d) Increment *i*.
  Let the final value of *i* be *N* (that is we have *N* run files)
- *Next step: merge the runs (next slide).....*

External Sort-Merge cont'd

Merge the runs (N-way merge). We assume (for now) that $N < M$.

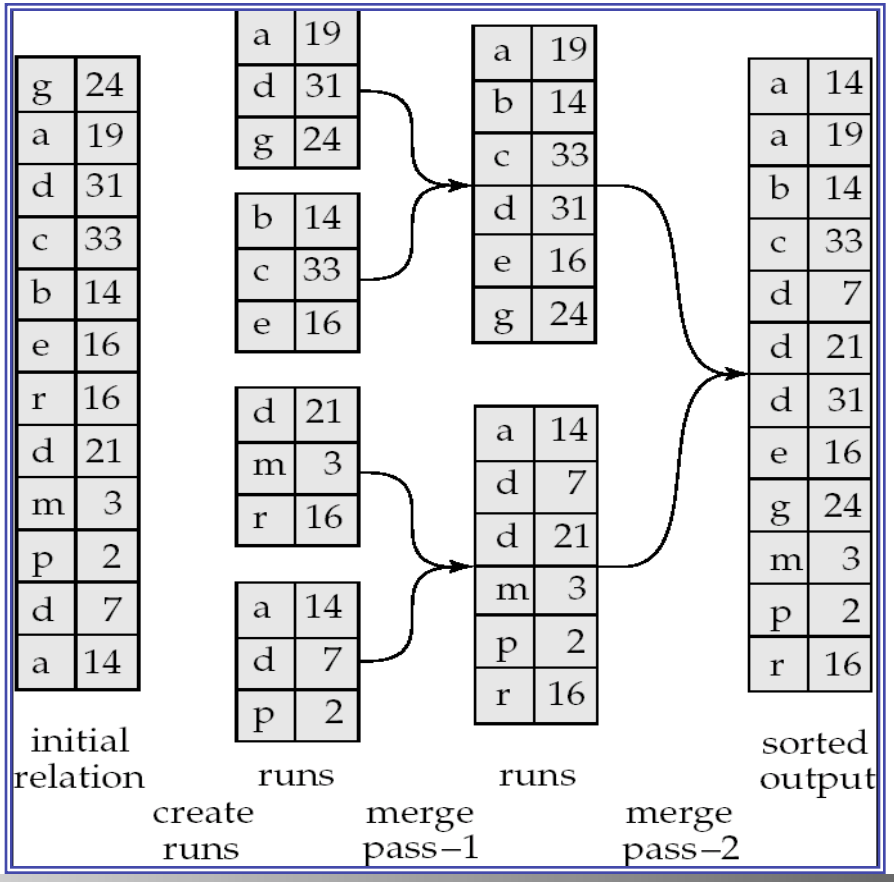
1. Use N blocks of memory to buffer input runs (1 for each of the N run files), and 1 block to buffer output. Read the first block of each run into memory.
2. **repeat**
   1. Select the first record in sort order (i.e., the smallest) among all buffer pages
   2. Write the record to the output buffer. If the output buffer is full write it to disk.
   3. Delete the record from its input buffer page.
      If the buffer page becomes empty **then**
      read the next block (if any) of the run into the buffer.
3. **until** all input buffer pages are empty

External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of $M - 1$ runs are merged.
  - A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor.
    - For example, if M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

Example: External Sorting Using Sort-Merge

Suppose that Memory holds at most three blocks, only one tuple fits in a block. The relation needs 12 blocks to store.



External Merge Sort (Cost analysis)

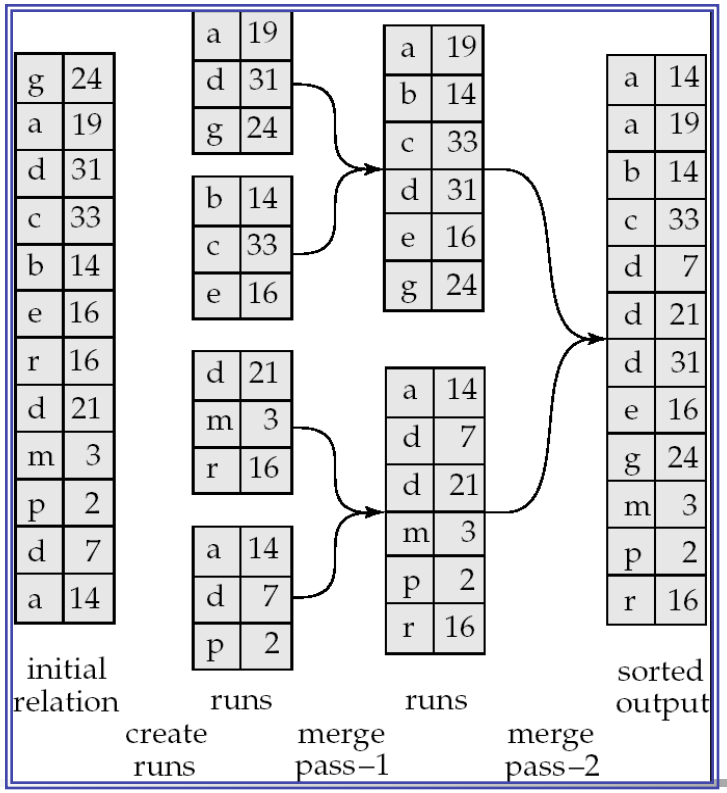
- Assume relation is stored in b_r blocks, M is the memory size, so the number of run file b_r/M.
- Buffer size b_b (read b_b blocks at a time from each run and b_b blocks for writing).
- Cost of Block Transfer
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$. Each time can merge $\lfloor (M-b_b)/b_b \rfloor$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$ (read/write all b_r blocks)
    - for final pass, we don't count write cost
    - Thus total number of block transfers for external sorting:

      $$2b_r + 2b_r \lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - b_r = b_r (2 \lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil + 1)$$
- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
    - Total number of seeks:

      $$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - 1)$$

Example: External Sorting Using Sort-Merge

Suppose that Memory hold at most three blocks, only one tuple fits in block. The relation needs 12 blocks to store, m= 3, *assume* b_b =1.



Total block transfer = ?

Total seeks = ?

End of Lecture

- Summary
  - Basic Steps in Query Processing
  - Query costs
  - Algorithms for evaluating relational algebra operations
    - Selection
    - Projection
    - Merge-sorting
- Reading
  - 6th edition, Chapters 12.1, 12.2., 12.3, 12.4
  - 7th edition, Chapters 15.1, 15.2., 15.3, 15.4

Database Development and Design (CPT201)

Lecture 4c:
Query Evaluation - Join

Dr. Wei Wang
Department of Computing

Learning Outcomes

- Algorithms for evaluating join operators
- Algorithms for evaluating other expressions

Natural-Join Operation

Notation: $r \bowtie s$

- Let r and s be relations on schemas R and S respectively. Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s
- Example:

  $R = (A, B, C, D)$

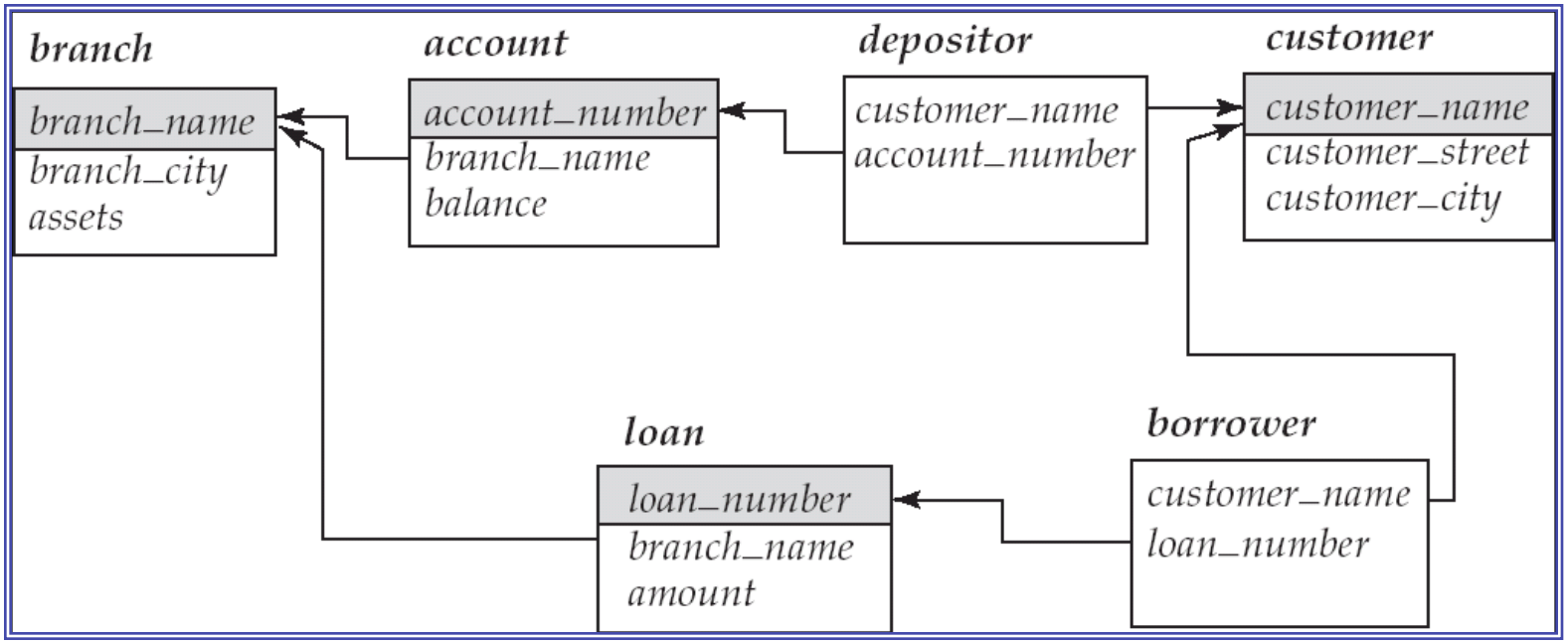
  $S = (E, B, D)$
  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:

    $$\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

Join Operation

- Several different algorithms to implement joins exist (not counting with the ones involving parallelism)
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- As for selection, the choice is based on cost estimate.

Banking Example



- Examples in next slides use the following information:
  - Number of records of *customer*: 10,000
  - Number of blocks of *customer*: 400
  - Number of records of *depositor*: 5,000
  - Number of blocks of *depositor*: 100

Nested-Loop Join

- The simplest join algorithms, that can be used independently of everything (like the linear search for selection)
- To compute the theta join: $r \bowtie_\theta s$

  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 θ
  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.
- Quite expensive in general, since it requires to examine every pair of tuples in the two relations.

Nested-Loop Join cont'd

- In the worst case, if there is enough memory only to hold one block of each relation, n_r is the number of tuples in relation r, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus
  $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- But in general, it is much better to have the smaller relation as the outer relation
- The choice of the inner and outer relation strongly depends on the estimate of the size of each relation.

Nested-Loop Join Cost in Example

- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5,000 * 400 + 100 = 2,000,100 block transfers,
    - 5,000 + 100 = 5,100 seeks
  - with *customer* as the outer relation
    - 10,000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks
- Instead of iterating over records, one could iterate over blocks. This way, instead of $n_r * b_s + b_r$ we would have $b_r * b_s + b_r$ block transfers
- This is the basis of the block nested-loops algorithm.

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 • t_s to the result.
      end
    end
  end
end
```

Block Nested-Loop Join Cost

- 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 (instead of once for each tuple in the outer relation).
- Best case (when smaller relation fits into memory): $b_r + b_s$ block transfers plus 2 seeks.
- Some improvements to nested loop and block nested loop algorithms can be made:
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer, reduce the number of disk access
  - Use index on inner relation (if available) to quickly get the tuples which match the tuple of the outer relation.

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - In some cases, it pays to construct an index just to compute a join.
- For each tuple t_r in the outer relation r, use the index on s to look up tuples in s that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of r, and, for each tuple in r, we perform an index lookup on s.
- Cost of the join: $b_r + n_r * c$ *block transfers and seeks*
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple in r
  - c can be estimated as cost of a single selection on s using the join condition (usually quite low, when compared to the join)
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Example of Indexed Nested-Loop Join Costs

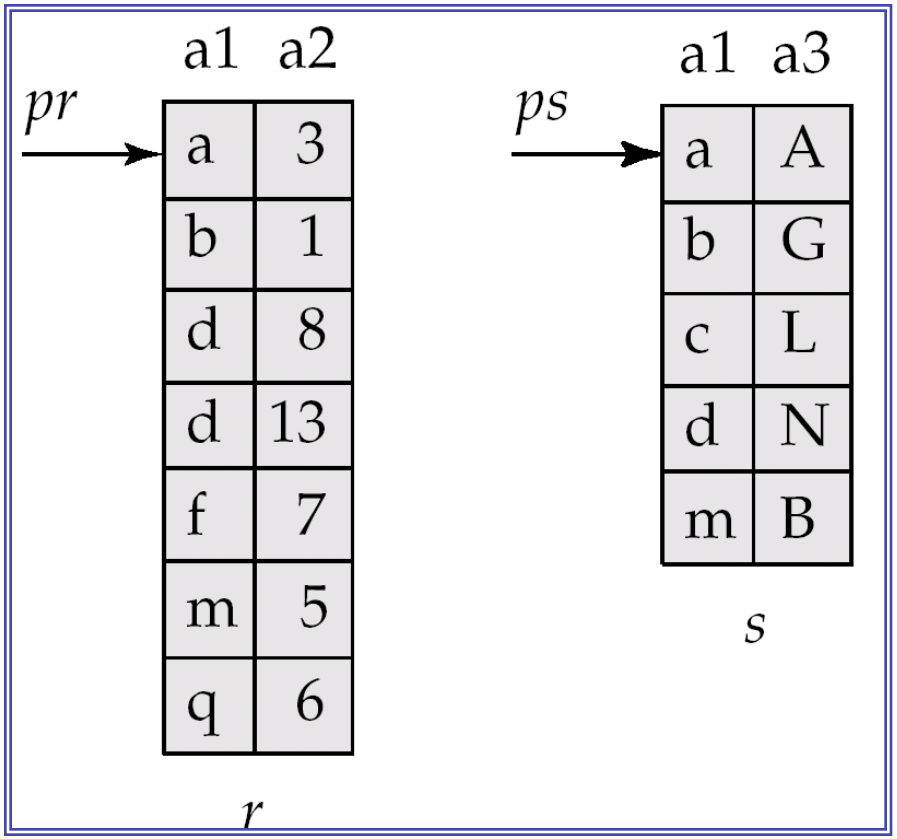
- Compute *depositor* ⋈ *customer*, with *depositor* as the outer relation.
- Let *customer* have a primary B⁺-tree index on the join attribute *customer-name*, with n=20.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5,000 tuples
- Nested loop join: 2,000,100 block transfers and 5,100 seeks
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
- Cost of indexed nested loops join
  - 100 + 5,000 * (4+1) = 25,100 block transfers and seeks.
  - The number of block transfers is less than that for block nested loops join
  - But number of seeks is much larger
  - In this case using the index doesn't pay (this is specially so because the relations are small)

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes). Join step is similar to the merge stage of the sort-merge algorithm.
2. Merge-join algorithm
    1. Initialise two pointers point to r and s
    2. While not done
        1. the pointers to r and s move through the relation.
        2. A group of tuples of inner relation s with the same value on the join attributes is read into S_s .
        3. Do join on tuple pointed by p_r and tuples in S_s;
    3. End while

Merge-Join cont'd

Read pseudocode in the textbook!



Merge-Join cont'd

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming that all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is (where b_b is the number of blocks in allocated in memory for each relation):

  $b_r + b_s$ block transfers +

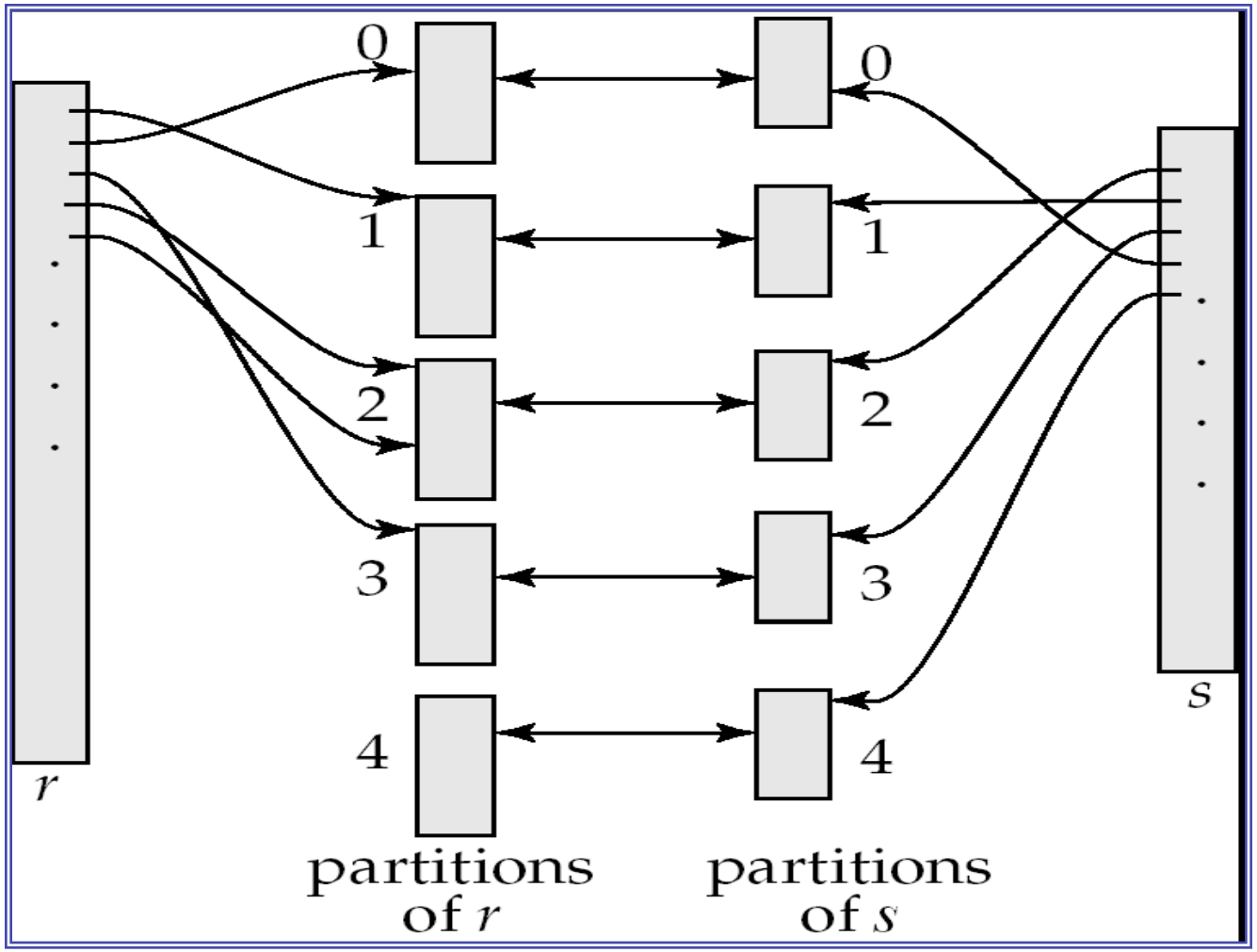
  $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

  - Plus the cost of sorting if relations are unsorted.
  - Since seeks are much more expensive than data transfer, it makes sense to allocate multiple buffer blocks to each relation, provided extra memory is available.

Hash-Join

- Also only applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.
- General idea:
  - Partition the relations according to this
  - Then perform the join on each partition r_i and s_i
    - There is no need to compute the join between different partitions since an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes. If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join cont'd



Hash-Join Algorithm

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, some blocks of memory (b_b) are reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i*:
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a ***different hash*** function than the earlier *h* for partitioning.
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and *r* is called the **probe input**.

Hash-Join algorithm cont'd

- The number of partitions *n* for the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2, to avoid overflows
  - The probe relation partitions r_i need not fit in memory

Cost of Hash-Join

- The cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers, and
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks
  - each of the n_h partitions could have a partially filled block that has to be written and read back
  - The build and probe phases require only one seek for each of the n_h partitions of each relation, since each partition can be read sequentially.
- If the entire build input can be kept in main memory (then no partitioning is required), Cost estimate goes down to $b_r + b_s$ *and 2 seeks.*

Cost of Hash-Join in Example

- For the running example, assume that memory size is 20 blocks $b_{depositor}$ = 100 and $b_{customer}$ = 400.
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass. Similarly, partition *customer* into five partitions, each of size 80. This is also done in one pass.
- Assuming 3 blocks are allocated for the input buffer and each output buffer
- Therefore total cost, ignoring cost of writing partially filled blocks:

  3(100 + 400) = 1, 500 block transfers +
  2(⌈100/3⌉ + ⌈400/3⌉) + 2*5 = 344 seeks

- We had up to here:
  - 40,100 block transfers plus 200 seeks (for block nested loop)
  - 25,100 block transfers and seeks (for index nested loop).

Other Operations: Duplicate Elimination

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimisation:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection:**
  - perform projection on each tuple;
  - followed by duplicate elimination.

Other Operations: Aggregation

- **Aggregation** can be implemented similarly to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimisation:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Other Operations: Set Operations

- **Set operations** (∪, ∩ and -): can either use variant of merge-join after sorting, or variant of hash-join.
- Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not in it.
          2. At the end, add the tuples in the hash index to the result.
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is in the hash index, delete it from the index.
          2. At the end, add remaining tuples in the hash index to the result.

End of Lecture

- Summary
  - Join
    - Nested-Loop Join
    - Block-Nested-Loop Join
    - Indexed-Nested-Loop Join
    - Sorted-Merge-Join
    - Hash Join
  - Other Operations
- Reading
  - 6th edition, Chapters 12.5 and 12.6
  - 7th edition, Chapters 15.5 and 15.6