

Chapter 1

Introduction

- Basic assumptions in databases
- System architecture for users
- Nature of databases
- Intractable problems in databases
- Coverage of topics in this book

1.1. The field of databases

- The basic assumption in the field of databases
 - Databases deal with storage and retrieval of *large* numbers of objects
 - Largeness implies occurrences of objects that share similar structures – thereby forming collections of similar objects
 - Thus a database is a set of collections of similar objects; the field mainly deal with the “large collections” aspect
 - Largeness necessitates stream-oriented processing of collections, leading to algebraic languages for query of data
- Query languages vs. general purpose languages (GPLs)
 - Query languages are much easier for application development when compared to GPLs such as Java
 - System can do dramatic optimization, largely relieving users from worries of runtime efficiency
 - Such a framework would be an illusive dream in a GPL
 - *This distinguishes databases from other programming systems*
- This book ...
 - ... undertakes a deeper understanding of query languages
 - ... considers several database paradigms
 - Relational: interesting in itself; helps us understand core concepts
 - Object-oriented: allows code and data reuse
 - XML: applies to a hugely diverse set of applications
 - ... helps us see that all query languages have a common theme
- Databases – as mini computer science
 - It has its own SQL-style (query) language(s)
 - Own techniques for security, concurrency, and recovery

1.2. Expectations of query languages

- Queries report subsets of existing data ...
 - ... in addition they may report some information about existing data
- They do not compute non-existing data
- What does this mean? We explain through an example
- Example: information about personnel in a company
 - John works in Toys and has a salary of 50K
Mary works in and manages Toys and has a salary of 60K
Leu works in Shoes and has a salary of 55K
 - We can draw following “subsets”
 - The names of employees are John, Mary, and Leu
 - Mary is John's manager
 - We can draw the following information
 - There are three employees
 - There is one manager
 - Mary has the highest salary
 - Average salary of employees in Toys is 55K
 - We cannot draw the following subsets
 - John works in Shoes
 - John manages Toys
 - John's address is 123 Main Street

1.3. Database applications

- A (specification of a) database application consists of
 - Entities that have an identifiable existence in the real world
 - Relationships among the entities
 - Constraints on entities and relationships
- States of an application database
 - A state consists of concrete entities and relationships
 - Constraints are to be satisfied by every state
- The objective is to query, analyze, and maintain data
- Example: the Personnel application
 - Entities are as follows
 - A Person, identified by his/her Name
 - An Employee, a Person, identified by Name, and has a Salary
 - A Department, identified by DName, the name of the Department
 - A Manager, an Employee, hence identified by Name
 - Relationships
 - An Employee works in a Department
 - A Department is managed by a Manager
 - Constraints
 - An Employee works in exactly one department
 - A department has at most one manager
 - (Some departments may not have a manager)

1.4. Mapping an application to a database system

- The database platform facilitates ...
 - Modeling a database schema
 - A storage model for data
 - A framework to help develop applications related to query, analysis, and maintenance of data
- Database schema design for an application
 - We are given an application and a (database) platform, e.g. Oracle
 - A database schema is designed
 - *This schema is* mapped to a database schema
 - Database schema is mapped to the platform
 - The mapping includes mapping of entities, relationships, and mechanisms to enforce constraints as database changes states
 - The database is then created and loaded reflecting the state of the application
 - The state of a database is said to be in a legal if it satisfies all constraints
 - Even though a database is dynamic, it must always remain in a legal state

1.5. Desirable features of a database system

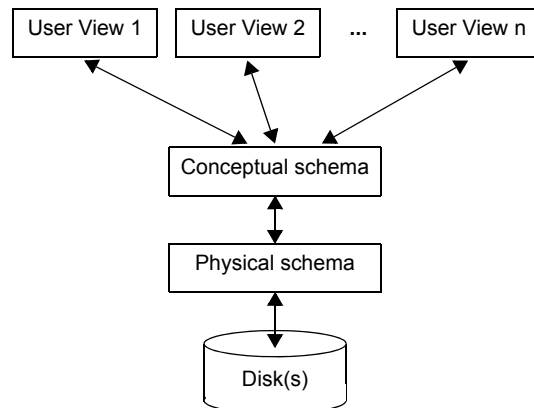
- Some properties are desirable in a database
 - Minimize, possibly eliminate, data redundancy
 - Must allow constraints to be defined and enforced
 - The information content of a database should mimic the application
- Minimization of redundancy
 - A data item that is identifiable uniquely should be stored once
For example, in Personnel application, detailed information about departments should not be repeated for every employee
 - This does not mean every value appears only once in a database
For an example, two employees may have the same salary by coincidence that may not hold in a different state of the database.
- Constraints and their enforcement
 - Database should allow constraints to be defined
 - For example, an employee may work in at most one department
 - Enforce constraints when the database is created and updated
 - It is desirable that such enforcement is efficient
- Content of information
 - In databases information associations are stored as well as computed
For example, if some part of the database associates $A = 5$ with $B = 6$, and another part associates $B = 6$ with $C = 7$, then the association of $A = 5$ with $C = 7$ may be inferred.

1.6. Database users

- Users of databases
 - DBA, database owner, application developer, end user
 - They interact with a database system in different ways
 - One or more persons could assume overlapping roles
- Database administrator (DBA)
 - Authorizes database owners to create databases
 - Overall in-charge of good health and performance of the dbms
 - In-charge of hardware needs, e.g. expanding storage by adding a disk
 - DBA's role could overlap with the overall system administrator
 - DBA may not have access to data residing in databases
- Database owner
 - A database owner creates a database
 - Authorizes users to access the database in specified ways
some users may even be authorized to enroll other users
 - Can fine tune performance, e.g., by creating indexes
- Application developer
 - Develops the database schema, storage strategies for data for fast access, programs, and user interfaces for an application
 - Help load data, start, and fine tune the application
 - Help in incremental changes and fine tuning
- End users
 - Authorized by the owner
 - Have access to specific parts of data for specific operations
 - May have authority to enroll other users

1.7. User view of system architecture

- The following shows the overall architecture



- Layered architecture: changes generally do not propagate upward
- Not all users interact directly with all layers
- Conceptual schema is also called the database schema
- The interaction of different users with the database system
 - DBA may expand, add, or reconfigure storage devices and change the physical schema to reflect the changes
 - An application developer does not directly interact with the database except as a consultant to help make technical decisions and expand functionality
 - Conceptual schema may be changed by a database owner
 - A database owner creates custom views for (communities) of users
 - Each view gives impression of a self contained database to end users
 - End users can query, maintain and interact with data to the extent authorized by the owner

1.8. Database paradigms

- Relational Database Paradigm - a springboard
 - Consists of a simple data model and query language (SQL)
 - SQL is user-friendly, declarative, leaving optimization to the system
 - SQL is algebraic and it has limitations
 - SQL is weaker than (any) general-purpose language (GPPL)
 - (We use Java as a place holder to refer to GPPLs)
 - Gap between Java and SQL is filled by a hybrid of SQL and GPPL
 - Java Database Connectivity (JDBC) is such a hybrid
 - All database paradigms mimic and improve the relational paradigm
- Object-orientated paradigm
 - Mimics and extends and refines relational database paradigm
 - Allows objects to have *nested* structures
 - *Inheritance* facilitates code reuse
 - *Referencing* facilitates data reuse by allowing data to reside elsewhere
 - Linguistics of queries becomes simpler for users
 - SQL reincarnates as OQL - object query language
- Semistructured paradigm (XML)
 - It extends relational and object-oriented paradigms
 - Covers data that is not uniformly structured
 - XML (extensible markup language) can be used to represent it
 - XML makes heterogeneous information seem homogeneous
 - SQL reincarnates as XQuery - XML query

1.9. Intractable problems in databases

- There are three intractable problems that are inherent in data
 - Impedance mismatch, rigidity of database schema, and null values
 - Some paradigms fair better than others
- Impedance mismatch
 - As algebraic languages such as SQL are weak they have to be mixed with general purpose language such as Java
 - Having to deal with algebraic and imperative features leads to *impedance mismatch* for users
- Rigidity of Database Structure
 - For a given database application, multiple schemas are possible
 - A schema, once chosen, seems to be cast in stone
 - Impedes evolution of reality reflected through data
 - Impedes integration of similar data with different schemas
 - XML eases the rigidity problem, but *does not* eliminate it
- Null values
 - Databases languages tend to handle bulk data uniformly
 - If salary is not defined, how would one answer “Salary > 100K?”
 - Nulls reflect uncertainty that propagates to queries
 - Many different semantics of a null value have been identified!
 - A study of incomplete information is a difficult area in databases
 - SQL allows users to use nulls at their own risk
- Application development has to circumvent above problems
 - These problems are inherent and deeply rooted in data
 - They *cannot* have satisfactory solutions in the field of databases

1.10. Our coverage of topics

- Paradigm centric approach to databases
 - Relational, object-oriented, and XML paradigms are covered
 - Each is a superset of the previous
 - All have SQL-like algebraic query languages
 - SQL, its variations and incarnations are covered, but not as dogma
 - We do not concentrate upon a specific platform, e.g. Oracle
- Efficiency issues
 - Architecture of system from performance point-of-view
 - Importance of minimizing page accesses and memory requirements
 - Algebraic query optimization, the backbone of databases
- Database schema design
 - Entity-relationship model for designing a database schema
 - Dependency theory, also for designing a database schema
- Java Database Connectivity (JDBC)
 - A hybrid of Java and SQL for relational databases
 - We show why efficiency should be of concern in JDBC programs
- Information integration
 - Sharing of information and bringing large information on line
 - Information integration to address the schema-rigidity problem
 - Query of summary information to help decision making

Chapter 2

The relational model

- Relations as formalization of tables
- Attributes
- Tuples
- Relation as a set of tuples
- Keys
- Constraints
(At present only covered under data dependencies)

2.1. Relational Model: Attribute and Schema

- A relation is formalization of a table
 - Ordering of rows or columns is disregarded
- Attributes
 - An attribute formalizes a column name
 - Definition: An attribute is an identifier and a type is associated with it.
- Examples
 - *Name* is an attribute of type *string*
 - *Salary* is an attribute of type *integer*
- Relation schema
 - Collects column names together
 - Definition: A *relation schema* is a set of attributes.
A relation schema is sometimes called a schema.
- Examples
 - {Name, DName, Salary} is a schema.
Here, DName is name of a department
- Notations for schemas
 - {Name, DName, Salary} can be written as “Name DName Salary”
or as “Name, DName, Salary”
 - The schema {A, B, C} can be written as “A B C” or “ABC”
 - Can be informal as long as no confusion arises in a given context

2.2. Relational Model: Tuples and Relations

- Tuple
 - Formalizes concept of a row or a record in a table
 - Definition: Suppose S is a schema. Then a tuple over S is an assignment of values to attributes of S . The value assigned to each attribute should be of the type associated with the attribute.
- Example
 - Suppose schema $S = \text{Name DName Salary}$ is given
Then the following is a tuple over S
(Name \rightarrow John, DName \rightarrow Toys, Salary \rightarrow 50K)
This tuple may be written as (John, Toys, 50K)
- Relation
 - Definition. Suppose S is a schema. A relation over S consists of finitely many tuples over S . (Can be empty, having 0 tuples.)
- Example
 - Consider a relation called “Emp” over Name DName Salary:
{(John, Toys, 50K), (Mary, Toys, 60K), (Leu, Shoes, 55K)}
 - We use 50K to informally represent 50,000
 - This relation can be displayed as:

Emp

Name	DName	Salary
John	Toys	50K
Mary	Toys	60K
Leu	Shoes	55K

2.3. Relational Model: Databases

- Relational database schema
 - Definition: A relational database schema consists of finitely many relation schemas.
 - For example, the following, called Personnel, is a database schema:
 $\{\text{Emp}(\text{Name}, \text{DName}, \text{Salary}), \text{Dept}(\text{DName}, \text{MName})\}$
- Relational database
 - A relational database over a relational database schema D consists of a relation for each relational schema in D.
- Example
 - The following is a relational database over the Personnel database schema.

Emp

Name	DName	Salary
John	Toys	50K
Mary	Toys	60K
Leu	Shoes	55K

Dept

DName	MName
Toys	Mary

2.4. Relational Model: Keys (3 slides)

- We are interested in the following specific types of keys
 - Superkey, key, primary key, and foreign key
 - All these types of keys are sets of attributes
- Superkey
 - A *superkey* is a set of attributes that identify tuples uniquely
 - Let's assume that two employees will never have the same name
 - Then {Name} is a superkey of Emp relation
 - This implies {Name, DName} is also a superkey of Emp
 - {Name, DName} & {Name, DName, Salary} are also superkeys of Emp
 - {DName} is not a superkey
- Key (note unqualified use of the term “key”)
 - A *key* is a minimal set of attributes that form a superkey
 - Only {Name} is a key of Emp, none of other three superkeys is a key
- A key can have multiple attributes
 - Consider enrollment relation that matches students and courses:
Enrollment(CourseCode, Section, StudentID, Grade)
 - A course is taken by multiple students, so CourseCode is not a key
 - A student takes multiple courses, so StudentID is not a key
 - However, {CourseCode, StudentID} is a key because for a given course there can be only one enrollment of a given student
 - Thus, a key can have multiple attributes
- A relation can have multiple keys
 - Give an abstract example of a relation $r(A,B,C,D,E)$ that has {A, B, C} as its key, as well as {B, D} as its key

2.5. Relational Model: Keys (slide 2 of 3)

- Primary key
 - This is more of a system concept than a logical one
 - A relation can have multiple keys
 - In SQL, a key can be designated as a *primary key*
 - This imposes an ordering of tuples in a relation; such ordering is not necessarily visible to a user
 - This implies that if there are multiple keys, only one can be designated as the primary key
 - This designation serves two purposes
 - Efficient retrieval: retrieval of a tuple with a given primary key value (e.g., Name = “John”) will be efficient
 - Enforcement of integrity: the system will not allow multiple tuples with the same (values of) primary key to reside in a relation
- Examples of primary key
 - {Name} can be designated as the primary key of Emp
 - Although a superkey such as {Name, DName} can be designated as a primary key, it does not seem wise as it is not minimal
 - {CourseCode, StudentID} can be primary key of enrollment
 - Only one of {A, B, C} as well as {B, D} can be the primary key of r
- Foreign key
 - Note that an enrollment record with a given StudentID does not make sense if no student has that StudentID
 - In SQL this can be done by designating StudentID as a foreign key
 - SQL requires a foreign key to be a primary key in some other relation
 - StudentID may be a primary key in Student (StudentID, Classification, GPA) relation

2.6. Relational Model: Keys (slide 3 of 3)

- There are many variations surrounding keys
 - *Unique* key vs. *nonunique* keys
 - *Minimal* keys vs. *superkeys*
 - *Primary* keys vs. *secondary* keys
 - *Local* keys vs. *foreign* keys
 - A “key”, without any qualifier, is unique, minimal, and local
 - A relation can have only one primary organization, but several secondary structures can be superimposed upon it
 - Secondary structures are far less efficient than primary organization
 - However, for certain kinds of accesses a secondary organization can be far more efficient than having no secondary organization

Chapter 3

Relational algebra

- Relational operators
- Algebra as a query language
- Covered in lectures:
 - Importance and meaning of operators
 - The five fundamental operators and Codd-completeness

3.1. Relational Algebra and SQL

- Why study relational algebra ...
 - Algebra can itself serve as a query language for relational databases
 - But its greater significance is that it forms the foundation for SQL
 - SQL, the structured Query Language, is the center-piece of databases
 - SQL is much easier to use when compared to general-purpose programming languages such as Java
- User vs. system in databases
 - Because of SQL's algebraic character ...
 - ... users express their queries with ease in SQL
 - ... system rewrites user queries in more optimal form for execution
 - This is *the* most fundamental and core idea in databases
- The centrality of SQL in databases, and this book
 - Query languages for all database models are inspired by SQL
 - In this book we consider OQL, the query language for objects, and XQuery, the query language for XML
 - There is a lot more to SQL than its syntax, e.g. query optimization
 - This book emphasizes deeper understanding of SQL
- Relational algebra consists of relational operators
 - Input consists of relation(s)
 - The output of an operator is also a relation
 - Relational operators can be composed to form relational expressions
 - Natural language queries can be expressed in relational algebra
- Relational operators are introduced next

3.2. Union, difference, and intersection operators

- These are defined if r and s have the same schema
 - $r \cup s = \{x: x \in r \vee x \in s\}$; its schema is same as that of r or s
 - $r - s = \{x: x \in r \wedge x \notin s\}$; its schema is same as that of r or s
 - $r \cap s = \{x: x \in r \wedge x \in s\}$; its schema is same as that of r or s
- Example

r

A	B
1	2
3	4
5	6
7	8

s

A	B
1	3
3	4
5	6

$r \cup s$

A	B
1	2
3	4
5	6
7	8
1	3

$r - s$

A	B
1	2
7	8

$r \cap s$

A	B
3	4
5	6

3.3. The selection operator

- $\sigma_c(r)$ selects those tuples of r that satisfy condition c
- Examples of conditions
 - $A \leq B$,
 - $A > 2 \wedge A < B$
- In $\sigma_c(r)$, c should only involve attributes of r .
- The schema of $\sigma_c(r)$ is same as that of r .
- $\sigma_c(r) = \{x: x \in r \wedge x \text{ satisfies condition } c\}$
- Example

r		$\sigma_{A > 2 \wedge A < B}(r)$	
A	B	A	B
1	2	3	4
3	4	5	6
5	6		
7	6		

- Note that the operator $\sigma_c(r)$ involves an implicit loop:
for tuple x in r
 if x satisfies condition c , then return x ;
- The syntax $\sigma_c(r)$ is user friendly

3.4. The projection operator

- $\Pi_X(r)$ retains only X columns
 - In $\Pi_X(r)$, X must consist of some attributes of r
- The schema of $\Pi_X(r)$ is X
- $\Pi_X(r) = \{x[X]: x \in r\}$
 - Here $x[X]$ includes only attributes in X, other attributes are excluded
- Example

r			$\Pi_{AC}(r)$	
A	B	C	A	C
1	2	4	1	4
5	4	2	5	2
5	6	2	7	3
7	8	3		

- $\Pi_{AC}(r)$ is a set from which a “duplicate tuple was removed”
- The schema of $\Pi_{AC}(r)$ is AC

3.5. The natural join operator

- $r \bowtie s$ combines tuples of r and s agreeing on common attributes
- $r \bowtie s$ is always defined
- Suppose schema of r is R and schema of s is S , Then schema of $r \bowtie s$ is $R \cup S$
- $r \bowtie s = \{x : x[R] \in r \wedge x[S] \in s\}$
 - For a tuple to be in $r \bowtie s$, its R portion should be in r and S portion in S
 - Implicitly, R and S portions match on the common attributes $R \cap S$
- Example

r		s		r \bowtie s		
A	B	B	C	A	B	C
1	2	2	3	1	2	3
3	4	4	4	3	4	4
5	6	4	6	3	4	6
9	4	9	4	9	4	4
10	5			9	4	6

- Natural join reflects the intuition that $B = 2$ in r represents the same thing as $B = 2$ in s
- What happens when schemas of r and s are disjoint?
 - Visit Appendix A if necessary
- What happens when schemas of r and s are the same?

3.6. The renaming operator

- Sometimes it becomes necessary to rename attributes or relations before or after an operator is applied
- Renaming does not alter the information content, i.e. tuples
- We use the syntactic form $\alpha \rightarrow \beta$ to express that the identifier α is changed to β .
- We do not care to formalize this
- Example
 - Consider $r \cup (s: C \rightarrow B): \rightarrow q$
 - Here, attribute C in s is renamed to B
 - After computing the union the result is named as q
 - “:” has the lowest priority
 - Here is an example

r		s		q	
A	B	A	C	A	B
1	2	1	3	1	2
3	4	3	4	3	4
5	6	5	6	5	6
				1	3

- Different authors use different notations
- Sometimes renaming is done automatically by the “system”

3.7. The cross product operator

- $r \times s$ concatenates tuples of r and s
- Every common attribute in r and s renamed
- In renaming prefix “r.” and “s.” are used to make attributes unique across r and s
- $r \times s = \{x \circ y : x \in \text{renamed } r \wedge y \in \text{renamed } s\}$
 - Here, \circ denotes concatenation of two tuples to form a larger tuple
 - The schema of $r \times s$ consists of all attributes of r and s after renaming
- Example

r		s		r × s			
A	B	B	C	A	r.B	s.B	C
1	2	4	2	1	2	4	2
3	4	7	3	1	2	7	3
5	6			3	4	4	2
				3	4	7	3
				5	6	4	2
				5	6	7	3

- Relations r and s are resident in $r \times s$. Why?
- When schemas of r and s are disjoint $r \bowtie s$ is $r \bowtie s$
When schemas of r and s are the same $r \bowtie s$ is $r \bowtie s$
- Some authors require schemas of r and s to be disjoint for $r \times s$ to be defined. In that case no implicit renaming is required.

3.8. Additional join operators

- A join merges the schemas and contents of operand relations
- We have already introduced $r \bowtie s$ and $r \times s$
- There are other join operators: left outer join, right outer join, and full outer join not covered here
- We end our coverage with $r \bowtie_c s$, the condition-join
- $r \bowtie_c s$ is a combination of cross product and selection
- It concatenates tuples of r and s and retains those satisfying condition c .
- Condition c involves attributes in r and s
- $r \bowtie_c s = \{x \text{ o } y : x \in \text{renamed } r \wedge y \in \text{renamed } s \wedge x \text{ o } y \text{ satisfies } c\}$
 - The schema of $r \bowtie_c s$ consists of all attributes of r and s after renaming
- Example

r		s		$r \bowtie_{r.B \neq s.B} s$			
A	B	B	C	A	r.B	s.B	C
1	2	4	2	1	2	4	2
3	4	4	3	1	2	4	3
5	6			5	6	4	2
				5	6	4	3

- $r \bowtie_c s$ can be defined in terms of other operators. How?

3.9. Natural Language Queries in relational Algebra

- Recall Personnel database

Emp

Name	DName	Salary
John	Toys	50K
Mary	Toys	60K
Leu	Shoes	55K

Dept

DName	MName
Toys	Mary

- Give names and salaries of employees in Toys or Credit.*

$\Pi_{\text{Name, Salary}}(\sigma_{\text{DName} = \text{'Toys'} \vee \text{DName} = \text{'Credit'}}(\text{Emp}))$

Name	Salary
John	50K
Mary	60K

- The query is independent of the state of the database
- Another algebraic expression for the query is as follows:

$\Pi_{\text{Name, Salary}}(\sigma_{\text{DName} = \text{'Toys'}}(\text{Emp})) \cup \Pi_{\text{Name, Salary}}(\sigma_{\text{DName} = \text{'Credit'}}(\text{Emp}))$

- Different users may express the query in different ways
- Who is John's manager?

$\Pi_{\text{MName}}(\sigma_{\text{emp.DName} = \text{Dept.DName} \wedge \text{Name} = \text{'John'}}(\text{Emp} \times \text{Dept}))$

- Observe how operators are composed
- Retrieves the following relation

MName
Mary

- Give a different expression that is more (most?) efficient.

Chapter 4

SQL: the Structured Query Language

- Query
 - The select statement,
 - Subqueries, union, except, intersection, exists, and all
- Creation of and changes to a database
- Views
- Keys and Integrity constraints
- Recommended
 - W3 School tutorial – <http://www.w3schools.com/sql/default.asp>
 - These tutorials are very accessible
 - Many other technologies are considered by W3 School

4.1. Select statement: Motivating example

- *Who is John's manager* in relational algebra:

$\Pi_{MName}(\sigma_{emp.DName=Dept.DName \wedge Name='John'}(Emp \times Dept))$

- In SQL it is expressed as follows

```
select d.MName
from Emp e, Dept d
where e.DName = d.DName
      and e.Name = "John"
```

- The from clause

- Sets up tuple variables e and d for Emp and Dept relations
- e and d are called aliases of Emp and Dept, respectively
- computes $Emp \times Dept$ a relation with 5 attributes

e.Name	e.DName	e.Salary	d.DName	d.MName

- The where clause applies the selection operator
- The select clause applies a projection and removes prefix “d.”

MName

4.2. Select statement: simple examples

- Recall Personnel database

Emp			Dept	
Name	DName	Salary	DName	MName

- List names and salaries of employees in Toys or shoes

```
select e.Name, e.Salary
from Emp e
where e.DName = 'Toys' or e.DName = "Shoes"
```

- SQL does not suppress duplicates by default

- Users can use distinct option

```
select distinct e.Salary, e.DName
from Emp e
where e.DName = 'Toys' or e.DName = "Shoes"
```

- If the result is to be reported in sorted order, use “order by”

```
select distinct e.Salary, e.DName
from Emp e
where e.DName = 'Toys' or e.DName = "Shoes"
order by e.Salary desc
```

- The desc option sorts tuples in descending order
- asc and desc must be used, not “ascending” and “descending”
- asc is the default, but it may be used for emphasis
- An order may be created for any number of attributes
For example: order by x.A asc, x.H desc, y.B

4.3. Select statement: aliasing

- A previous SQL query could also be written without aliases

```
select Dept.MName
from Emp, Dept
where Emp.DName = Dept.DName and Name = "John"
```

- The prefix “Emp.” and “Dept.” are used for clarity or to remove ambiguity
 - In Emp.DName = Dept.DName the use of both prefixes is required
 - No prefix is used in Name = "John"; could use Emp.Name = "John"
 - Emp stands for a relation as well as a variable
 - It turns out that aliases cannot be eliminated
- Consider: List pairs of names of employees such that the two employees work in the same department.

```
select e1.Name as Name1, e2.Name as Name2
from Emp e1, Emp e2
where e1.DName = e2.DName
      and e1.Name < e2.Name
```

- Retrieves

Name1	Name2

- In the above query e1 and e2 are aliases of Emp
 - There is only one Emp relation
 - e1 and e2 are independent variables that range over all tuples in Emp
- How would you do it without (variable) aliasing?
 - The Paradox story

4.4. Aggregates functions

- SQL supports the following aggregate functions
 - sum, avg, count, min, and max
 - Each computes a value for a *group of tuples*
- The following syntactic forms are allowed
 - sum ([[all] | distinct] expression), the avg function has similar syntax
 - count ([[all] | distinct] expression) or count(*)
 - max(expression), the min function has similar syntax

- Consider the following example

```
select sum(all e.Salary)
from Emp e
where e.DName = 'Toys'
```

- Here, all “Toys” tuples are treated as a single group
sum(e.Salary) computes sum of all salaries in the group
- Variations
 - With the all option, the keyword all is (always) redundant
Thus, sum(all e.Salary) is usually written as sum(e.Salary)
 - sum(distinct e.Salary) returns the sum of distinct salary-values
 - avg(e.Salary) computes average of all salaries in the group
 - avg(distinct e.Salary) returns the average of distinct salary-values
 - count(e.Salary) returns count of tuples in the group
 - count(*) also returns count of tuples in the group
 - Among the above two examples, the latter seems more natural
 - count(distinct e.Salary) returns count of distinct salaries
 - min(e.Salary) returns minimum salary-value in the group of tuples
 - min(distinct e.Salary) is not available, perhaps because it is not useful

4.5. Select statement: full form

- Syntax

```
select [distinct] selectList
from fromList
[where tupleCondition]
[group by attributeList]
[having groupCondition]]
[order by attributeList]
```

- select and from clauses are required
- select clause has a distinct option
- Where clause is optional
- group by and having clauses are optional.

We can have group by without the having clause

We cannot have having clause without group by clause

- order by clause is optional

- Semantics

- The query is a filter applied to a database
- It only reports information, does not manufacture it (Java does)
- Every clause acts like a filter, making SQL query easy to understand
from → where → group by → having → select → order by
- There is no long memory, only output of a step is used in next

- Group by and having clauses

- After where clause all tuples are considered to form a single group
If group by is used, this group gets further partitioned
- Aggregates are applied to each group
- where clause is used to express a condition to filter tuples
- having clause is used to express a condition to filter groups of tuples

4.6. Using aggregate functions

- List names of departments and total salary of privileged employees in the department, for those departments that have at least 10 privileged employees. A privileged employee is one whose salary is at least 100K.

```
select e.DName, sum(e.Salary) as SumSal
from Emp e
where e.Salary >= 100000
group by DName
having count(*) >= 10
order by SumSal desc
```

- from clause filters out all relations except Emp
- where clause filters out tuples of employees that are not privileged
- group by clause groups these tuples in groups – one per department
- having clause discards groups (departments) that have fewer than 10 privileged employees.
- For each group, the select clause reports aggregate and other values. It also removes the nesting of groups.
- The order by clause orders the result in decreasing order of SumSal values
- Retrieves

Name	SumSal

- What if for each department we want total salary...
 - ... of “all” employees instead of “privileged employees”
- The two scenarios have the same grammatical structure in English
- But the SQL query for “all employees” has a more complex structure

4.7. More syntactic forms

- Syntactic forms allowing nesting
 - A in (L)
 - A not in (L)
 - A op (L)
 - A not op (L)
 - A op any (L)
 - A op all (L)
 - exists (L)
 - not exists (L)
- In addition we have
 - (select ...) union (select ...)
 - (select ...) except (select ...) corresponding to difference
 - (select ...) intersection (select ...)

4.8. Using “in” and union

- List names of employees managed by Mary or Lin

```
select e.Name
from Emp e, Dept d
where e.DName = d.DName
      and (d.MName = "Mary" or d.MName = "Lin")
```

- The following uses “A in (L)”

```
select e.Name
from Emp e
where e.DName in ( select d.DName
                  from Dept d
                  where d.MName = "Mary"
                  or d.MName = "Lin")
```

- The following uses union

```
( select e.Name
  from Emp e, Dept d
  where e.DName = d.DName and d.MName = "Mary")
union
( select e.Name
  from Emp e, Dept d
  where e.DName = d.DName and d.MName = "Lin")
```

- The following uses union in the inner query

```
select e.Name
from Emp e
where e.DName in ( ( select d.DName
                  from Dept d
                  where d.MName = "Mary" )
                union
                ( select d.DName
                  from Dept d
                  where d.MName = "Lin" ) )
```

4.9. Using “not in” is tricky!

- List names of employees who have a manager

```
select e.Name
from Emp e
where e.DName in (select d.DName from Dept d)
```

- The following will also work

```
select e.Name
from Emp e, Dept d
where e.DName = d.DName
```

- List names of employees who do not have a manager

```
select e.Name
from Emp e
where e.DName not in (select d.DName from Dept d)
```

- The following will not work

```
select e.Name
from Emp e, Dept d
where e.DName != d.DName
```

4.10. The keyword all, any, A op (L), and coercion

- Names of employees with highest salaries in their departments

```
select e.Name
from Emp e
where e.Salary >= all ( select e1.Salary
                        from Emp e1
                        where e1.DName = e.DName)
```

- The following will also work

```
select e.Name
from Emp e
where e.Salary = ( select max(e1.Salary)
                  from Emp e1
                  where e1.DName = e.DName)
```

The inner relation coerced to an ordinary value

- Consider the variable **e** in each subquery above
 - The variable **e**, used in subquery is defined in outer query
 - This makes the subquery correlated
 - Subqueries in previous examples are not correlated

- What will the following return?

```
select e.Name
from Emp e
where e.DName = "Toys"
and e.Salary >= any (select e1.Salary
                     from Emp e1
                     where e1.DName = "Shoes")
```

- This will return names of every employee in Toys whose salary is greater than some employee in Shoes

4.11. Constructs exists and any, A op (L) and coercion

- Consider the following query invoking coercion
 - It tries to query Names of employees having colleagues earning at least 100K
- ```
select e.Name
from Emp e
where e.DName = (select e1.DName
 from Emp e1
 where e1.Salary >= 100000)
```
- SQL will not complain about the syntax of this query  
At run time the system will try to coerce the inner relation to a scalar
  - If inner query results in a single department, the query will work, otherwise there will be runtime error!
  - Thus it may work for some states of the database and not for others
- The following three will work and will not give syntax error

```
select e.Name
from Emp e
where e.DName = any (select e1.DName
 from Emp e1
 where e1.Salary >= 100000)
```

```
select e.Name
from Emp e
where e.DName in (select e1.DName
 from Emp e1
 where e1.Salary >= 100000)
```

```
select e.Name
from Emp e
where exists (select e1.*
 from Emp e1
 where e1.Dname = e.DName
 and e1.Salary >= 100000)
```



## 4.12. Codd-completeness of SQL retrieval operators

- There are five basic algebra operators (plus renaming)
  - union, difference, selection, projection, and join
- A language that has expressive power of these operators is called Codd-complete
- The select statement captures selection, projection and join  
This is why it is called an spj-expression
- In addition we have
  - (select ...) union (select ...) corresponding to union
  - (select ...) except (select ...) corresponding to difference
- Therefore, SQL is Codd-complete
- Java is obviously Codd-complete.
  - Why?
  - Actually Java is far more powerful than just being Codd-complete
- Minimally Codd-complete languages are important because
  - They can be made user-friendly
  - They admit superb optimization
- Optimization of SQL is superb
- Java can be only marginally optimized

## 4.13. More on SQL

- The distinct option
  - Removes duplicates
  - One would expect duplicates to be removed by default
  - Unfortunately, this is not so
- Nulls
  - Null values are allowed
  - $A \text{ op } B$  results in “unknown” if either A-value or B-value is null
  - “unknown or true” is true, “unknown and false” is false.
  - Only tuple for which the where clause becomes true are retrieved
  - Nulls give rise to all kinds of problems and can retrieve results that are counter intuitive
  - A study of incomplete information is a full fledged and difficult area in databases
  - Fourteen different semantics of a null value have been identified!
- Syntax of SQL
  - Historically ad hoc, but improvements have been made
  - In a *functional language* if it seems logical to use some language construct in some place the language must allow it
  - Unfortunately, SQL is not fully functional

## 4.14. SQL: Creation of a database

- Recall Personnel database

| Emp  |       |        | Dept  |       |
|------|-------|--------|-------|-------|
| Name | DName | Salary | DName | MName |
|      |       |        |       |       |

- We do not consider GUI's, privileges, and system catalog here
- To create a db execute from operating system prompt ">"  
> createdb Personnel
- This creates a database with no tables in it.
- To use this database
  - The database should be loaded. The prompt changes to "SQL>"  
> loaddb Personnel
  - Then execute commands  
SQL> \_
- Commands for creating Emp and Dept relations

```
SQL> create table Emp (
 Name char (30),
 DName char (10),
 Salary integer)
```

```
SQL> create table Dept (
 DName char (10),
 MName char (30))
```

## 4.15. SQL: Modifying a table R

- Insertion, updates, and deletions: the syntax

- insert into R [(Attribute-list)]  
values (value-list)
- update R  
set <value assignments>  
where <condition>
- delete from R  
where condition

- Examples

- insert into Emp  
values ('Hari', "Credit", 45000);
- insert into Emp (Salary, Name, DName)  
values (45000, "Hari", "Credit");
- insert into Emp (Name, DName, Salary)  
values ("Hari", Null, 45000);
- insert into Emp (Name, Salary)  
values ("Hari", 45000);
- update Emp  
set Salary = Salary + 1000, DName = "Shoes"  
where DName = "Toys"
- delete from Emp  
where Salary > 1000000  
and DName = "Accounting"

## 4.16. Views in SQL

- A query stands for a relation, yet to be materialized

```
create view empDept (Name, DName) as
 select e.Name, e.DName
 from Emp e
```

- It defines empDept “relation” with Name and DName attributes
  - A view can be optimized, compiled and stored, ready for execution
  - Views can be customized to a user’s need
  - Views can be used to hide information, Salary in this case
- Views and relations can be made available to a user
    - For example {empDept, Dept}. The user can query his/her database; e.g. who is John’s manager?

```
select d.MName
 from empDept e, Dept d
 where e.Dname = d.DName and e.Name = "John"
```
    - The system will rewrite the query directly in terms of stored relations, and optimize, compile and execute it.

Can empDept view be updated? What does this mean?

- Base relations have to be updated so the recomputed views appear to be updated.
  - empDept can be updated
- Not all views can be updated in arbitrary ways

#### 4.17. A view that cannot be updated

- Consider a database with following relations and the view

| r |   | s |   |
|---|---|---|---|
| A | B | B | C |
| 1 | 2 | 2 | 4 |
| 3 | 2 | 2 | 5 |

create view J (A, C) as  
select x.A, y.C  
from r x, s y  
where x.B = y.B

- The view stands for the following virtual relation:

| J |   |
|---|---|
| A | C |
| 1 | 4 |
| 1 | 5 |
| 3 | 4 |
| 3 | 5 |

- We cannot delete the tuple (3,4)  
We cannot update (3,4) to (6,4)
- “When a view can or cannot be modified?” can be complex
- SQL provides some conservative solutions

## 4.18. Designation of keys in SQL

- Keys have been discussed under Relational Model
- Relations can be designated to have primary and foreign keys
  - Designation of primary or foreign keys is not required
  - One primary and several foreign keys can be designated
  - A foreign key has to be a primary key in another relation that we term foreign relation. (Some systems, e.g. MySQL, may not enforce this.)
  - To designate a foreign key the keywords “foreign key” or “references” are used
  - If a foreign key consists of only one attribute, in order to designate it as the foreign key, the option of using the keyword “references” in its declaration can be invoked

- Example

```
create table Enrollment(
 CourseCode char(6) NOT NULL,
 Section int NOT NULL,
 StudentID char(9) NOT NULL references Student,
 Grade char(4),
 primary key (CourseCode, StudentID),
 foreign key (CourseCode, Section)
 references Offering (CourseCode, Section))
check ((select count (*)
 from Enrollment e
 group by e.CourseCode, e.Section) <= 25);
```

- About foreign keys ...
  - A foreign key ensures that a tuple cannot be created unless a *foreign tuple* with the given primary key exists in the foreign relation
  - What if a foreign tuple is deleted in future?
  - Deletions can cascade, raising very complex issues in the real world
  - SQL allows elaborate provisions to facilitate them

## 4.19. Integrity constraints in SQL

- SQL has elaborate provisions for enforcement of integrity
- Primary keys enforce uniqueness when tuples are inserted
- Foreign keys also enforce integrity, but they are intricate
  - What happens to a tuple when its corresponding tuple is deleted in the other relation?
  - SQL provides a cascading provision for deletions
- How to restrict attribute values?
  - include "check (Salary >= 0 and salary <= 100000)" in create Emp
  - Salary type can be customized and then be used in create Emp
  - A named constraint can be declared and used in multiple creates
- How to restrict number of tuples?

```
check ((select count (*)
 from Enrollment e
 group by e.CourseCode, e.Section) <= 25)
```

  - This constrain could be added to create Enrollment statement
  - What does it do?
  - Note the versatility of queries in SQL!
- Integrity constraints are triggered when updating a database
- Sophisticated, user defined triggers can be associated with update statements in SQL



## 4.20. Miscellaneous: Access control and catalog in SQL

- The database owner can control access to
  - Relations and views
  - Database operations
- grant and revoke commands are used
- Even a grant privilege can be given or revoked
- Examples:  
grant insert, delete on Enrollment to Jane  
revoke delete on Enrollment to Jane  
grant select to Hari on Enrollment
- This is quite elaborate in SQL
  - Users can give privileges to other users
  - This can cause unexpected behavior
- User actions can be audited by creating an audit trail
- Audit trails are relations that can be queried
- A system catalog is organized as a database
- Access to catalog can also be controlled
- Catalog can also be queried in SQL

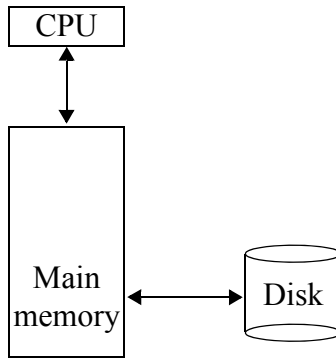
## Chapter 5

### *Physical organization*

- Disk vs. main memory
  - Pages and buffers
- Physical storage of relations
- Performance metrics
- Operations in a database and their cost
- Physical organization and indexing
  - Ordinary organization
  - Sorted files
  - B+ trees
  - Hashing

## 5.1. Physical storage: pages and buffers

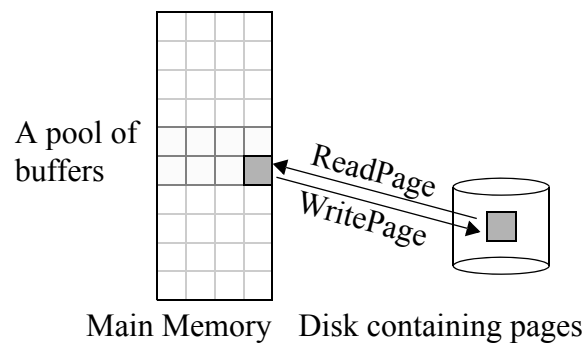
- Usual view of a computer system



- Processing is done in CPU, which is very fast
- Information to be processed must be in main memory
- Traffic between CPU and main memory is very fast (in nano-seconds)
- Traffic between main memory and disk is very slow (in milliseconds)
- If information is on the disk, this can create a bottleneck
  - To keep CPU busy, we must transfer (read/write) large chunks at a time between disk and main memory
- Gives rise to concepts of page and buffer
  - A page on the disk consists of a few kilobytes
  - A buffer in main memory has same size as a page
  - A read operation copies contents of a page into a buffer in memory; write does just the reverse

## 5.2. A model for performance in databases

- Assumptions
  - A database is very large and it is stored on the disk
  - Data needs very simple processing
  - Therefore: page accesses govern the throughput
- Database view of a computer system



- Notation
  - Number of pages in a relation:  $N$
  - Number of tuples in a relation:  $n$
- Performance metrics
  - The number of page accesses (fewer is better): most important
  - Number of buffers (fewer is better): also very important
  - Amount of disk space (less is better): less important

### 5.3. Logical Relation vs. Physical Relation

- At logical level, Emp relation is a set of tuples

| Name | ID  | DName | Salary |
|------|-----|-------|--------|
| Anna | 151 | 54    | 30K    |
| Art  | 306 | 52    | 30K    |
| Ben  | 200 | 32    | 45K    |
| Chi  | 310 | 36    | 42K    |
| Chin | 101 | 26    | 40K    |
| Don  | 311 | 15    | 30K    |
| Joe  | 312 | 30    | 28K    |
| John | 305 | 40    | 30K    |
| Mary | 165 | 25    | 23K    |
| Paul | 160 | 15    | 52K    |
| Pete | 300 | 38    | 32K    |
| Roy  | 180 | 28    | 47K    |

- At physical level, Emp relation is a sequence of pages

|                       |                      |                       |                       |                       |                       |
|-----------------------|----------------------|-----------------------|-----------------------|-----------------------|-----------------------|
| Anna; 151;<br>54; 30K | Ben; 200;<br>32; 45K | Chin; 101;<br>26; 40K | John; 305;<br>40; 30K | Paul; 160;<br>15; 52K | Pete; 300;<br>38; 32K |
| Art; 306;<br>52; 30K  | Chi; 310<br>36; 42K  | Don; 311;<br>15; 30K  | Mary; 165;<br>25; 23K |                       | Roy; 180;<br>28; 47K  |
|                       |                      | Joe; 312;<br>30; 28K  |                       |                       |                       |

0

1

2

3

4

5

- Currently, it uses 6 pages (N=6) to store 12 tuples (n = 12)  
A page can hold up to 3 tuples  
Space utilization is 67% (12 of 18 slots are being used)

## 5.4. Operations and performance: Examples

- Assume that the Emp relation is sorted by Name
  - It consists of  $n$  records stored in  $N$  pages
- 1. Report John's salary
  - Use binary search
  - Cost:  $\lceil \log_2 N \rceil$  page accesses; 1 buffer
- 2. Find who works in department 15? Ignore cost of writing.
  - Do a linear scan of pages
  - Cost:  $N$  page accesses; 1 buffer
- 3. Insert a new record.
  - Binary search for place of insertion. If page has space for a record, add the new record in its buffer. Write buffer to disk.
  - Cost:  $\lceil \log_2 N \rceil + 1$ ; 1 buffer. What if the page is full?
- 4. Give a raise to an employee.
  - Binary search for the record, change the salary value, and write the page containing the record back.
  - Cost:  $\lceil \log_2 N \rceil + 1$ ; 1 buffer.
- 5. Delete an employee's record.
  - Binary search, remove from buffer, write buffer to disk.
  - Cost =  $\lceil \log_2 N \rceil + 1$ ; 1 buffer. Problem: cannot repeat too often.
- 6. Correct the name of an employee.
  - Deletion followed by insertion.
  - Cost:  $2 * (\lceil \log_2 N \rceil + 1)$ ; 1 buffer

## 5.5. File structures and indexes for fast access

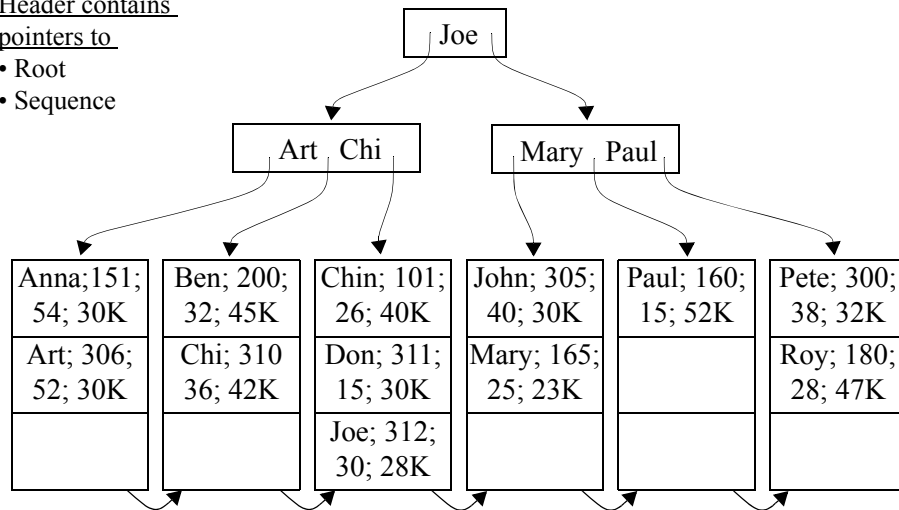
- The Emp file in our example is sorted by name
- This allows binary search; too slow for databases
- Database files can be organized for much better performance
- In databases we are ...
  - Not limiting ourselves to performance in terms of big-O, e.g.  $O(\log N)$
  - ... mindful of every page access being made
- B+ trees and hashing are most common, discussed next

## 5.6. B+ tree

- A B+ tree consists of an index and a sequence
  - Organized as a search tree supporting a key; nodes are disk pages
  - The *index* provides access paths to tuples by their key-values
  - The sequence consists of pages containing tuples
  - The tuples in the sequence are sorted by their key values
  - All nodes in the sequence are at the same level  
Therefore a B+ tree is perfectly balanced
  - Every node (except possibly the root) is at least half full.  
This guarantees a good fanout, small height, excellent performance
- Example: Emp organized as a B+-tree
  - Height of this tree is 3
  - Retrieval by key values require 3 page accesses
  - It can be traversed sequentially

Header contains  
pointers to

- Root
- Sequence





## 5.7. B+ trees: Rules of thumb

- Size (Number of pages) and space utilization
  - To facilitate random access, every node is a page on the disk
  - Every node (except possibly the root) is at least half full.
  - The capacity of an index node typically ranges from 50 to 200
  - Minimum fanout of an index node with a capacity of 100 keys is 51
  - This guarantees a large fanout; small height; good performance
  - Typically, height of a B+tree with 1 million records may 3 to 5
- B+ tree can support
  - Primary unique key
  - Secondary unique and non-unique keys
  - In case of a B+tree supporting a non-unique key, multiple records with a given key value may occupy several pages
- B+ tree performance
  - Suppose the key is K, height is h, and fanout is large (50 to 200 or so)
  - Retrieval of a record by K-value requires h page accesses and 1 buffer
  - Sequential scan by K-values: N page accesses  
(N = number of nodes in the sequence)
  - Range scan by K-values  $a \leq K \leq b$ : (h-1) + number of pages covered by the range (h-1 page accesses before going to the page containing K = a, followed by a sequential traversal until K > b)
  - Most of the time, an insertion requires h+1 page accesses and 2 buffers (to facilitate node splits)
  - Mark a record for deletion: h+1 page accesses

## 5.8. Hashing: Rules of thumb

- Hashing attempts to compute page-address of a record from its key value
  - The computation succeeds most of the time
  - Thus most records can be retrieved by a single page access
  - Some records require additional page accesses
- Hashing performance (approximate)
  - Retrieval of a record by K-value: Close to 1 page access, say 1.25
  - Insertion: 2.5 page accesses
  - Delete a record: 2.5 page accesses
- Hashing vs. B+ trees
  - For retrieval of single records by their keys hashing is more efficient
  - B+ trees facilitate sequential access, hashing does not
  - B+ trees also support range searches
  - Overall, B+ trees are more versatile

## 5.9. Primary and secondary physical organizations

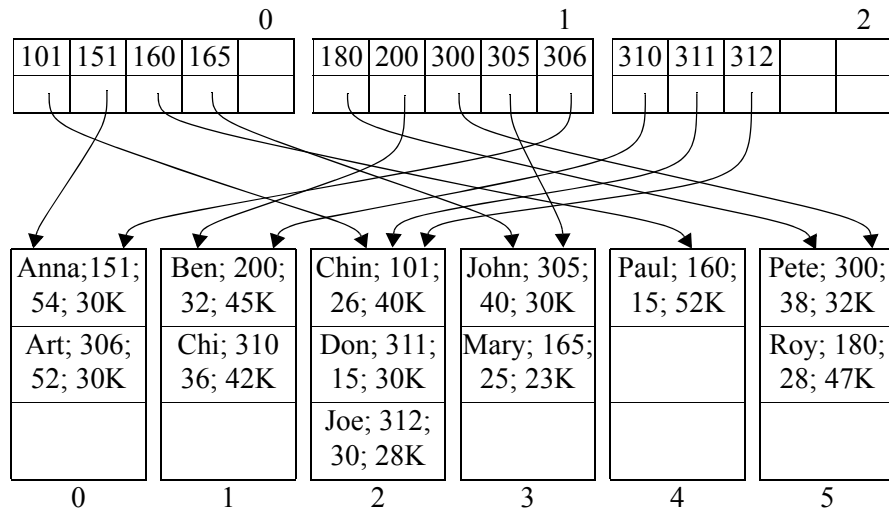
- Primary physical organization
  - A primary physical organization fixes the location of records  
Page 56 shows Emp organized as *sorted by Name-values*  
Page 59 shows Emp organized as a B+ tree
  - ID is another unique key of the Emp relation  
But neither Emp, on Pages 56 or 59, is organized by ID-values
  - If we sort Emp by IDs it will no longer remain sorted by Name  
A similar remark can be made about B+ tree
  - Obviously, we can have only one primary organization
- How can we provide faster access by IDs?
  - Instead of relocating records just point to them  
Then organize the pointers
  - Such an organization is called a secondary organization
- Primary and secondary keys
  - For Emp physical file on Page 56 Name is the *primary key*
  - Let's propose ID as a *secondary key*
- Suppose  $x$  is a record and  $k$  is its key, primary or secondary
  - Then  $x$  is also denoted as  $\bar{k}$  and  $\#x$  denotes the page address of  $x$
  - Note that the page address of  $x$  is also given as  $\#\bar{k}$
  - For Emp physical file on Page 56:  
 $\overline{\text{John}}$  as well as  $\overline{305}$  is the record (John, 305, 40, 30K)  
 $\#\overline{\text{John}}$  as well as  $\#\overline{305}$  is the page address 3
- Secondary records
  - Secondary records are pairs  $(k, \#\bar{k})$
  - Secondary organization organizes these secondary records

## 5.10. Example of a secondary organization for Emp

- A secondary organization for Emp for IDs
  - For ID as the key the secondary records are (151, #151), (306, #306), (200, #200), (310, #310), (101, #101), (311, #311), (312, #312), (305, #305), (165, #165), (106, #160), (300, #300), (180, #180).
  - This is same as (151, 0), (306, 0), (200, 1), (310, 1), (101, 2), (311, 2), (312, 2), (305, 3), (165, 3), (106, 4), (300, 5), (180, 5).
  - After sorting they are (101, 2), (151, 0), (160, 4), (165, 4), (180, 5), (200, 1), (300, 5), (305, 3), (306, 0), (310, 1), (311, 2), (312, 2).
- Suppose the secondary organization is also a sorted file
  - Assume that a secondary page can hold up to is 5 secondary records.
  - The following shows the desired physical organization

| 0   |     |     |     |  | 1   |     |     |     |     | 2   |     |     |  |  |
|-----|-----|-----|-----|--|-----|-----|-----|-----|-----|-----|-----|-----|--|--|
| 101 | 151 | 160 | 165 |  | 180 | 200 | 300 | 305 | 306 | 310 | 311 | 312 |  |  |
| 2   | 0   | 4   | 3   |  | 5   | 1   | 5   | 3   | 0   | 1   | 2   | 2   |  |  |

- The following shows both the primary and secondary organizations  
Secondary index on emp file by ID

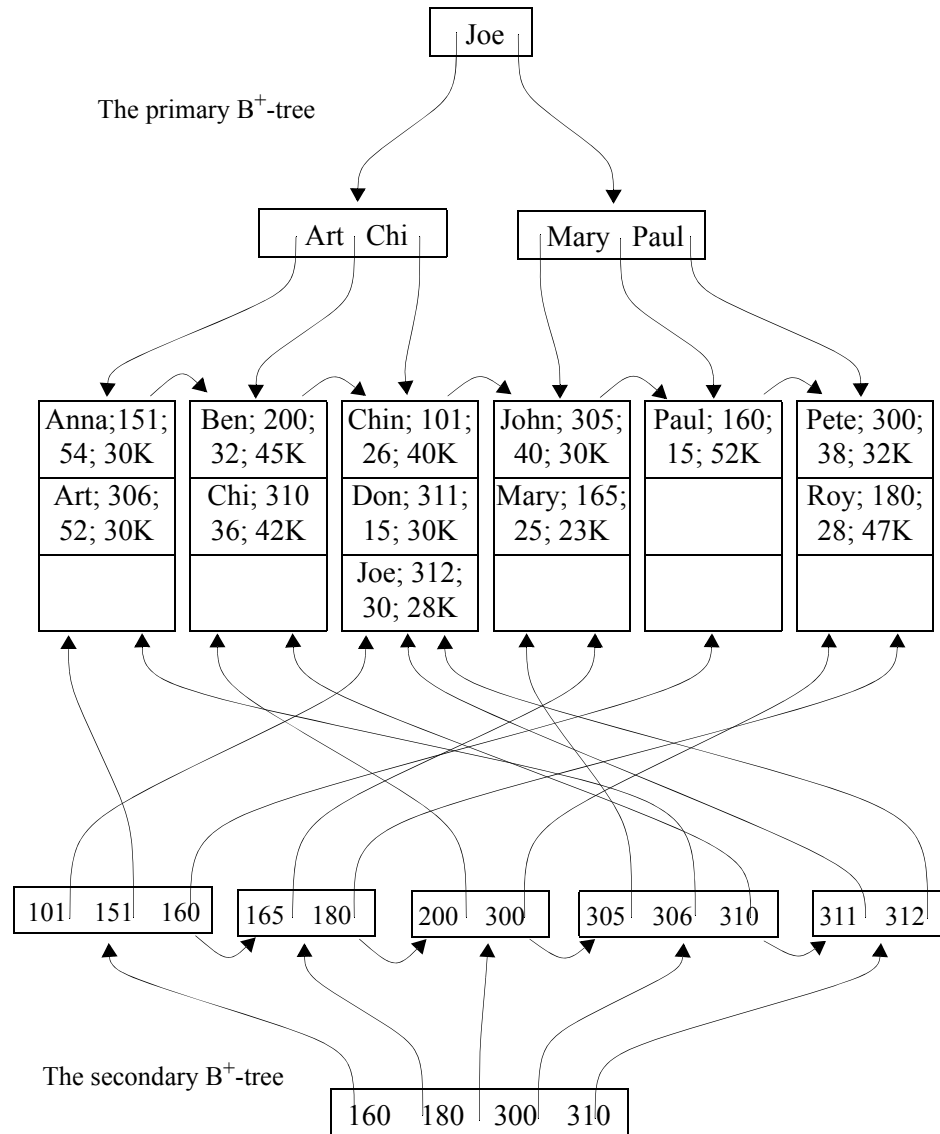


### 5.11. Performance of primary and secondary organizations

- Assume the following for primary and secondary files
  - Number of primary and secondary records are  $n$  and  $m$ , respectively
  - Number of primary and secondary pages are  $N$  and  $M$ , respectively
- Retrieval of a (primary) record by secondary key-value  $k$ 
  - First visit the secondary structure to locate the pair  $(k, \# \bar{k})$
  - Make an additional page access to retrieve the page with address  $\# \bar{k}$
  - Example for the Emp file  
For the ID-value 305, fetch the secondary record  $(305, \# \overline{305})$   
The cost of finding  $(305, \# \overline{305}) = \lceil \log_2 3 \rceil = 2$  page accesses  
The secondary record  $(305, \# \overline{305})$  is  $(305, 2)$   
Cost of reading primary page at address 2 is 1 page access
- Traversal records in Emp in order of ID-values
  - We can do this because secondary records are sorted by ID-values
  - This can also be done for B+ trees for IDs, but not for linear hashing
  - Scan through the secondary pages: the cost is  $M$
  - Follow each secondary record to fetch the primary record: the cost is  $n$
  - Thus the cost of traversal is  $M + n$
  - This much higher than  $N$ , the cost of traversal by primary key
  - The cost of traversal for Emp by primary and secondary keys are  $N = 6$  and  $M + n = 3 + 12 = 15$  page accesses, respectively
- An organizing technique can be used as primary or secondary
  - For sorting it has been shown on Page 63
  - Next slide on Page 65 shows this for B+ trees

## 5.12. Example: primary and secondary B<sup>+</sup> trees for Emp

- Primary and secondary B<sup>+</sup> trees for Emp are shown below



## Chapter 6

### *Query processing*

- Query processing is stream-based
  - An input stream is a conduit for pages from the disk to main memory  
An output stream is a conduit for pages from the main memory to disk
  - Selection has 1 input stream and 1 output stream  
Natural join has 2 input streams and 1 output stream
  - Every stream is supported by one or more buffers of its own
  - Performance is measured in numbers of page accesses and buffers.  
We disregard processing time and physical contiguity of pages on the disk. These are important issues, but our assumptions help us gain good initial understanding of query optimization without distractions
  - We consider the average time to access a random page as a unit of time
- Algebraic optimization is a core concept in databases
  - Algebraic framework is pivotal to success of databases: users write their queries easily and system rewrites them more optimally
  - Several plans may be considered and their costs estimated based on info in database catalog. (A plan is a template for an algorithm)
  - The winning plan is compiled and executed
  - Implementation of optimizer is estimated to take 40-50 person-years of work in commercial systems!
  - Excellent optimization is possible because SQL is weaker than general purpose programming languages such as Java
  - We will consider algebraic optimization of simple select-from-where statement of SQL

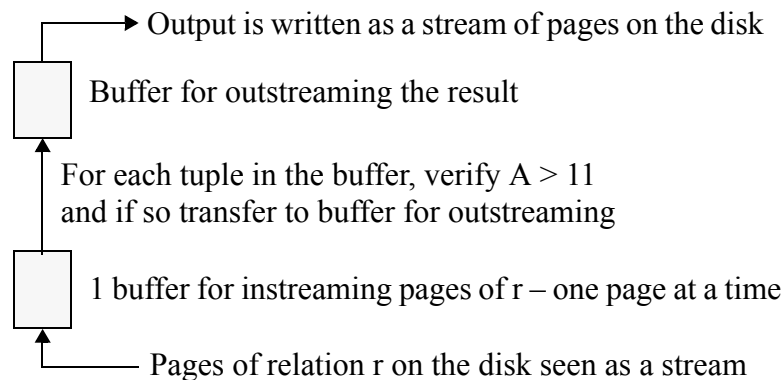
## A list of concepts in this chapter

- Basic concept of a stream (of pages)
  - Selection operator on a single relation (see Page 68)  
It uses 1 buffer for instreaming, 1 for outstreaming
- The rest of this page deals with query of two relations
  - Either  $r \bowtie s$  or Query( $r, s$ ) – an SQL query with a from clause with two relations (see Page 76 for a running example)
- Basics
  - Natural join with 3 buffers (see Page 71)
  - Using more buffers to reduce cost of natural join (see Page 72)
  - Algebraic optimization (see Page 78)
  - Baseline combines algebraic optimization and use of available buffers
- Consider strategies that try to improve baseline
- Some observations
  - Cost of output is same for all strategies (see Pages 77 and 78)
  - Query( $r, s$ ) costs no more than  $r \bowtie s$  because all operations other than  $\bowtie$  can be done on the fly without additional disk accesses (see Page 76)
  - When join attributes form the key of one of the relations the number of joining tuples is linear (see Page 75)
- Strategies considered in this book
  - Preprocessing operands of join (see Pages 78 and 79)
  - Index nested loop use index by join attributes (see Pages 73 and 81)
  - Sort merge if join attributes form a key on one of the relations (see Pages 74 and 80)



## 6.1. Selection Operator: a simple example

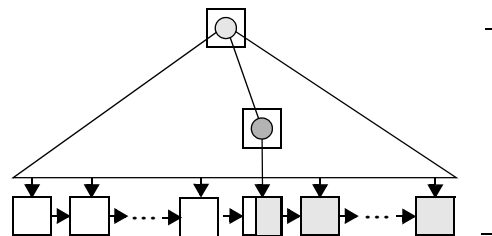
- Query to be processed:  $\sigma_{A>11}(r)$ 
  - The data resides on the disk; the output is also to be stored on the disk
- Catalog info
  - Assume that  $r$  has 200 pages and 40% of the tuples of  $r$  satisfy  $A > 11$
- Input and output streams and buffer allocation
  - There is 1 input and 1 output stream; each stream is allocated 1 buffer
  - The buffer attached to input stream is used to read one page at a time from disk to main memory; the buffer attached to output stream is used to write one page at a time from main memory to disk



- Strategy
  - Input 1 page of  $r$  at a time. For each tuple ask if  $x.A > 11$ . If not, reject the tuple. If yes, copy the tuple to the output buffer
  - Write output buffer whenever it becomes full
- Cost
  - Input  $r$ : 200 page accesses (because  $r$  consists of 200 pages)
  - Output  $200 * 0.40 = 80$  page accesses (Only 40% tuples are written)
  - Total: 280 page accesses; 2 buffers used during 280 units of time

## 6.2. Selection operator: using B+ tree to retrieve a range

- Query to be processed:  $\sigma_{A > 11}(r)$
- Catalog info
  - Assume that  $r$  occupies 200 pages and 40% of the tuples satisfy  $A > 11$
  - Assume that we have a B+tree with  $A$ -values as key (B+ tree preexists – not created on-the-fly for the query)



- Given
  - Height is 3
  - Records reside in 200 leaf pages.

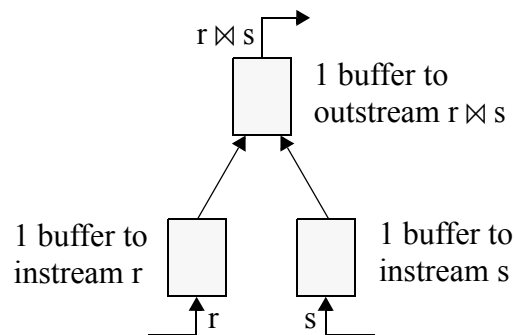
- Input and output streams and buffer allocation
  - Same as last example: 1 buffer for each stream
- Strategy
  - Same as before except that using the buffer for input stream, with 2 page accesses reach the first page containing records with  $A > 11$ .
  - All records satisfying  $x.A > 11$  are in this page and pages following it. These records will cover  $200 * 0.40 = 80$  pages
- Cost
  - Input  $r$ :  $2 + 80 = 82$  page accesses using 1 buffer
  - Output: 80 page accesses using 1 buffer
  - Total cost: 162 page accesses and 2 buffer
  - Relative to previous strategy, it saves  $(280 - 162)/280 = 42\%$
- Possible improvement
  - Input and output streams could share the same buffer.

### 6.3. Selection operator: using an index to match a key attribute

- Query to be processed:  $\sigma_{B=20}(s)$ 
  - We will not consider the cost of output this time
  - Assume that the matching tuple will be kept in memory
- Catalog info
  - The relation  $s$  occupies 400 pages on the disk
  - B-values form a key of the relation  $s$
  - We have a preexisting index on  $s$  by B values.  
We will consider two different cases of index:
    - > Case 1: B+tree index with performance of 3 page accesses
    - > Case 2: Hash index with an average performance of 1.25 page accesses
- Input and output streams and buffer allocation
  - 1 buffer for input stream
  - There is no output stream; we will leave the result in main memory
- Strategy
  - Use the index to lookup the record satisfying  $B = 20$
  - We are done as the cost of output is not to be considered
- Cost
  - Case 1: B-tree: 3 page accesses; 1 buffer
  - Case 2: Hash index: On average 1.25 page access; 1 buffer

## 6.4. Natural join: a basic nested-loop join strategy

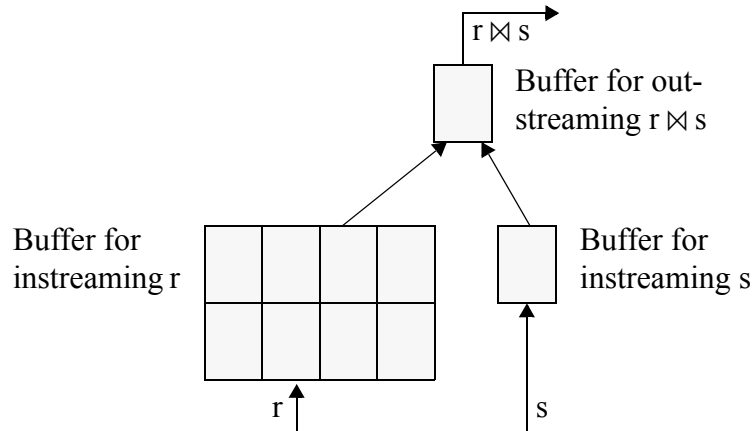
- Query to be processed:  $r \bowtie s$ , natural join of  $r$  and  $s$
- Catalog info
  - $r$  has 200 pages,  $s$  has 400 pages, and  $r \bowtie s$  occupies 250 pages
- Input and output streams and buffer allocation
  - There are 2 input streams and 1 output stream  
Use 1 buffer to instream  $r$ , 1 to instream  $s$ , and 1 to outstream



- Strategy
  - Every pair of pages from  $r$  and  $s$  should be in main memory some time
  - Read pages of  $r$  and  $s$  in a nested loop;  $r$  as outer and  $s$  as inner relation. For each pair of pages from  $r$  and  $s$  in memory, examine all pairs of tuples. Join matching tuples and deposit them in the output buffer.
  - Write the output buffer every time it becomes full
- Cost
  - Input  $r$ : 200 page accesses to read  $r$
  - Input  $s$ : read 400 pages of  $s$  200 times at  $400 * 200$  page accesses
  - Output: write 250 pages of the join relation (given)
  - Total =  $200 + (400 * 200) + 250 = 80,450$  page accesses
  - Key observation: Most time is spent in instreaming the inner relation

## 6.5. Natural join: using more buffers when available

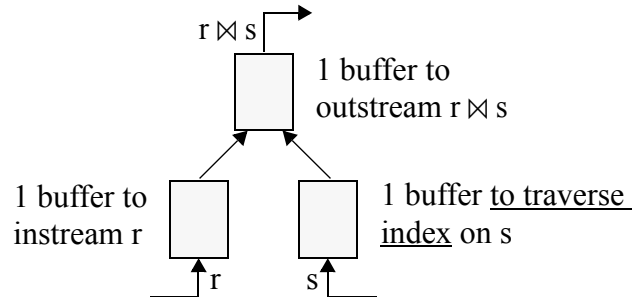
- Query to be processed:  $r \bowtie s$ , natural join of  $r$  and  $s$ 
  - Assume that up to 10 buffers are available
- Catalog info
  - $r$  has 200 pages,  $s$  has 400 pages, and  $r \bowtie s$  occupies 250 pages
- Input and output streams and buffer allocation
  - Use 1 buffer for inner relation  $s$  and 1 buffer for output
  - Use remaining 8 buffers for instreaming outer relation  $r$



- Strategy
  - Every pair of pages from  $r$  and  $s$  should be in main memory some time
  - Input  $r$ , filling all 8 buffers at a time. For each 8-page segment of  $r$ , read every page of  $s$  once. Rest is the same as before.
- Cost
  - The only change:  $s$  not instreamed 200 times, only  $\lceil 200/8 \rceil = 25$  times.
  - Previous cost =  $200 + (400 * 200) + 250 = 80,450$  page accesses  
 New cost =  $200 + (400 * \lceil 200/8 \rceil) + 250 = 10,450$  page accesses
  - Wise use of buffers saves us 87% page accesses!

## 6.6. Natural join with index-nested loop

- Compute  $r \bowtie s$ 
  - Assume that up to 10 buffers are available
- Catalog info
  - $r$  has 200 pages and 8,000 tuples,  $s$  has 400, and  $r \bowtie s$  needs 250 pages
  - Join attribute is B and it is key in one of the relations, say  $s$ ; assume that B+tree of height 3 is available on  $s$
- Input and output streams, buffer allocation, and strategy



- Buffer allocation: Use 1 buffer each for input  $r$ , input  $s$ , and output.
- Read 1 page of  $r$  at a time. For each tuple of  $r$ , consider its B-value and use the index to find a tuple in  $s$  with the matching B-value. If a match is found, compute the join tuple and deposit it to the output buffer
- Cost
  - Input  $r$ : 200 page accesses
  - Input  $s$  assuming a B+tree: 8000 \* 3 page accesses
  - Output 250 page accesses
  - Total for B+tree  $200 + 8000 * 3 + 250 = 24,450$  page accesses
- What if we have a hash index instead of B-tree?
  - Same strategy. Replace  $8000 * 3$  by  $8000 * 1.25$  (here, 1.25 is assumed to be the average cost for consulting the has index)

## 6.7. Natural join: sort-merge (sort relations and merge)

- Given relation  $r$  and  $s$ ; compute  $r \bowtie s$ 
  - Up to 10 buffers are available
  - Sorting  $M$  pages with  $k$  buffers requires  $2M * \lceil \log_{k-1} M \rceil$  page accesses ( $k - 1$  buffers for instreaming, 1 for outstreaming;  $\log_{k-1} M$  passes are needed; a pass requires  $M$  pages to be read and  $M$  pages to be written)
- Catalog info
  - Given:  $r$  has 200 pages,  $s$  has 400 pages, and  $r \bowtie s$  occupies 250 pages  
The join attribute,  $B$ , is key in (at least) one of the relations, say  $s$
- Motivation for sorting
  - Because  $B$  is the key of  $s$ , a tuple of  $r$  joins with at most one tuple of  $s$ . If  $r$  and  $s$  are sorted, join of  $r$  and  $s$  can be done by “merging” them; merging avoids having to read “inner” relation multiple times!
- Input and output streams, buffer allocation, and strategy
  - Sort  $r$  and  $s$  using all 10 buffers ( $k = 10$ ); then merge the two relations to perform the natural join
- Cost
  - Sort  $r$ :  $2 * 200 * \lceil \log_9 200 \rceil = 2 * 200 * 3 = 1,200$  page accesses; 10 buffers
  - Sort  $s$ :  $2 * 400 * \lceil \log_9 400 \rceil = 2 * 400 * 3 = 2,400$  page accesses; 10 buffers
  - Join  $r$  and  $s$  by merging:  $200 + 400 = 600$  page accesses; 3 buffers
  - Total: 4,200 page accesses; 10 buffers during 3600 and 3 buffers during 600 units of time. (Cost of output has been ignored.)
- Observations
  - The sorted copies require  $200 + 400$  pages on the disk for sorted copies. These pages are returned back to the storage after completion of join.
  - If a relation is sequence set of a B-tree on join attribute it is already sorted.

## 6.8. A running example for processing of SQL query

- Available resource: 10 buffers to process queries
- Given two relations *r* and *s* with catalog info as follows
  - Non-italicized information is given; *italicized* information is deduced

|                               | Relation <i>r</i>                                                             | Relation <i>s</i> |
|-------------------------------|-------------------------------------------------------------------------------|-------------------|
| Attributes                    | A, <u>B</u> , C                                                               | <u>B</u> , D, E   |
| Key                           | A                                                                             | B                 |
| Common (join) attribute       | B is in <i>r</i> as well as <i>s</i> ; note that <u>it is key in <i>s</i></u> |                   |
| Attribute Sizes in bytes      | A  =  B  =  D  = 8,  C  =  E  = 4                                             |                   |
| <i>Tuple size</i>             | <i>20 Bytes</i>                                                               | <i>20 Bytes</i>   |
| Page size                     | 1000 Bytes available for tuples                                               |                   |
| Capacity of a page            | <i>50 tuples</i>                                                              | <i>50 tuples</i>  |
| Space utilization             | <u>80%</u>                                                                    | <u>80%</u>        |
| <i>Average tuples / page</i>  | <i>40 tuples</i>                                                              | <i>40 tuples</i>  |
| Total number of tuples        | 8,000                                                                         | 16,000            |
| <i>Total size of relation</i> | <i>200 pages</i>                                                              | <i>400 pages</i>  |
| File structure                | Assumed as needed for illustration                                            |                   |

- We consider the following SQL query for next few slides

```

select x.A, x.C, y.D
from r x, s y
where x.B = y.B
 and x.A > 11
 and y.E = 55

```

- Join attribute is the key of *s*. Therefore, a tuple in *r* joins with at most one tuple in *s*; *r* ⋈ *s* has no more than 8,000 tuples – each consisting of 32 bytes. In the worst case, *r* ⋈ *s* occupies approximately 250 pages.



## 6.9. SQL query and its expression tree

- Given relations  $r(A,B,C)$  and  $s(B,D,E)$  and the SQL query

```
select x.A, x.C, y.D
from r x, s y
where x.B = y.B
 and x.A > 11
 and y.E = 55
```

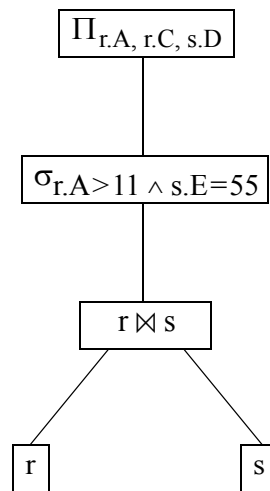
- Translation to relational algebra and representation as a tree

$$\Pi_{r.A, r.C, s.D}(\sigma_{r.B=s.B \wedge r.A>11 \wedge s.E=55} (r \times s))$$

- Replacement of cross product with natural join

$$\Pi_{r.A, r.C, s.D}(\sigma_{r.A>11 \wedge s.E=55} (r \bowtie s))$$

- Representation as expression tree



- General guide for evaluation strategy
  - Proceed to perform the natural join using any strategy
  - Do selection and projections on-the-fly while the tuples are in buffers
  - While processing, deposit only surviving join tuples to output buffer

## 6.10. SQL query: a simple plan

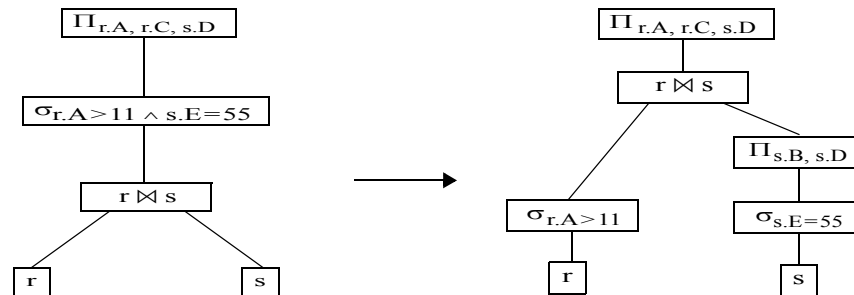
- Given: relations  $r(A,B,C)$  and  $s(B,D,E)$ , the SQL query

```
select x.A, x.C, y.D
from r x, s y
where x.B = y.B
 and x.A > 11
 and y.E = 55
```

  - Size of a page is 1,000 bytes; up to 10 buffers are available
- Catalog info
  - $r$  has 200 pages,  $s$  has 400 pages, and  $r \bowtie s$  occupies 250 pages
  - $r.A > 11$  is satisfied by 40% tuples of  $r$  and  $y.E = 55$  by 4% tuples in  $s$
- Strategy
  - We proceed to evaluate  $r \bowtie s$  with any strategy
  - For each pair of joining tuples in buffers apply selection & projection on-the-fly and transfer only surviving information to the output buffer
  - Compare with cost of  $r \bowtie s$ : no change in cost of input; output reduces
- Estimate the size of the output
  - Recall that 8,000 pairs of tuples join (see Page 75)
  - $A > 11$  and  $s.E = 55$  reduce this number to 40% and 4%, respectively  
Size of an  $(x.A, x.C, y.D)$  tuple in the select clause is 20 bytes
  - With 1,000 byte pages, output has  $\lceil 8000 * 0.40 * 0.04 * 20 / 1000 \rceil = 3$  pages, not 250. Cost of every strategy reduces by 247 page accesses
- Cost
  - As an example consider cost of sort-merge: 7,200 page accesses; 10 buffers during 3600 and 3 buffers during 3600 units of time.
  - This becomes: 6,953 ( $7200 - 247$ ) page accesses; 10 buffers during 3,600 and 3 buffers during 3600  $-247 = 3,353$  units of time.

## 6.11. SQL query – algebraic optimization

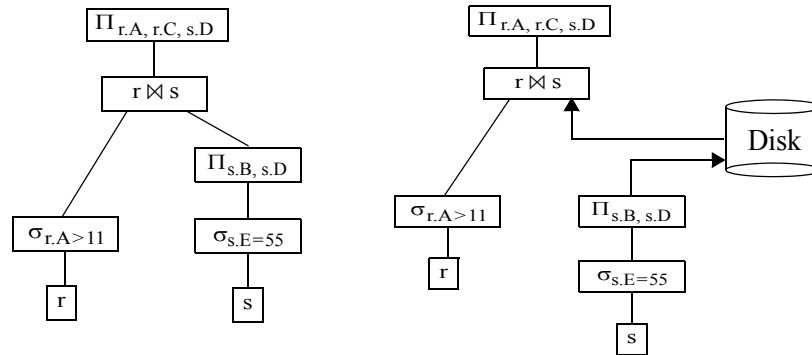
- Draw more efficient expression tree using algebraic identities



- How does algebraic optimization help?
  - Often, preprocessing can reduce sizes of one or both operands of  $r \bowtie s$   
This can reduce the cost of the join operator.
  - If we were to preprocess  $r$   
 Compute  $\sigma_{r.A > 11}(r)$  and write to disk.  
 Cost of pre-processing = 200 + 80 = 280 page accesses; 2 buffers  
Gain: the relevant portion of  $r$  for  $r \bowtie s$  reduces from 200 to 80 pages.
  - If we were to preprocess  $s$   
 Compute  $\Pi_{s.B, s.D}(\sigma_{s.E = 55}(s))$  and write to disk  
 Size of output:  $\sigma_{s.E = 55}$  reduces the size to 4%,  $\Pi_{s.B, s.D}$  reduces each 20 byte tuple to 16 bytes. Output =  $400 * 0.04 * (16/20) = 13$  pages  
 Cost of pre-processing = 400 + 13 = 413 page accesses; 2 buffers  
Gain: the relevant portion for  $s$  for  $r \bowtie s$  reduces from 400 to 13 pages.
- Which operands to preprocess?
  - This depends on buffer availability, sizes of relations, the expected amount of reduction in sizes, and the plan under consideration
  - Next, we will consider the nested loop join. In that case we will preprocess (only)  $s$ , leading to 91.90% savings in page accesses.

## 6.12. SQL query: algebraic optimization+preprocessing

- Draw more efficient expression tree using algebraic identities



- Strategy

- Preprocess  $s$ , write the smaller size to disk, then compute  $r \bowtie s$  and during this while tuples are in buffers, apply other operators on-the-fly
- Cost of pre-processing =  $400 + 13 = 413$  page accesses (see Page 78)
- Gain: the relevant portion of  $s$  has reduced from 400 to 13 pages  
With  $s$  as outer relation,  $r$  will have to be instreamed fewer times

- Cost

- Preprocessing of  $s$ : 413 page accesses; 2 buffers
- Computation of query:  $13 + 200 * \lceil 13/8 \rceil + 3 = 416$  page accesses; 10 buffers (recall estimated size of output as 3 pages – see Page 77)
- Total:  $413 + 416 = 829$  page accesses; 2 and 10 buffers

- Observations

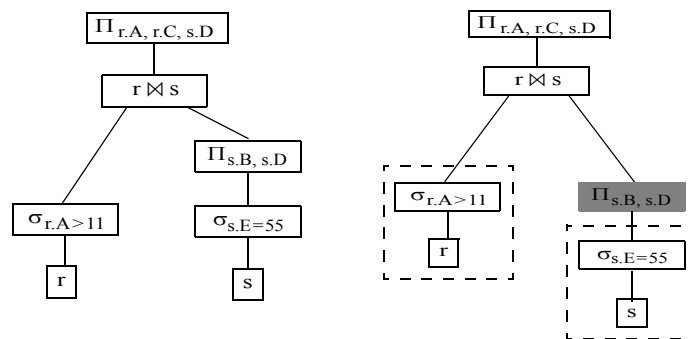
- This shows the importance of algebraic underpinnings of SQL
- A well implemented dbms will always do good optimization
- Question to consider: Why did we preprocess only  $s$ ? For the given query, think of scenarios when it will be beneficial to preprocess only  $r$ , both  $r$  and  $s$ , and none of  $r$  and  $s$

## 6.13. SQL query: algebraic optimization + sort-merge

- Recall
  - Because B is the key of s, a tuple of r joins with at most one tuple of s. If r and s are sorted, join of r and s can be done by “merging” them
  - With preprocessing, only relevant portions of r and s need to be sorted
  - Relevant portion of r and s have 80 and 13 pages (not 200 and 400)
- Cost
  - Preprocess r:  $200 + (200 * 0.40) = 280$  page accesses; 2 buffers (see Page 78)
  - Preprocess s: 413 page accesses; 2 buffers (see Page 78)
  - Sort r:  $2 * 80 * \lceil \log_9 80 \rceil = 2 * 80 * 2 = 320$  page accesses; 10 buffers
  - Sort s:  $2 * 13 * \lceil \log_9 13 \rceil = 2 * 13 * 2 = 52$  page accesses; 10 buffers
  - Join by merging:  $80 + 13 + 3$  (output) = 96 page accesses; 3 buffers
  - Total:  $280 + 413 + 320 + 52 + 96 = 1161$  page accesses

## 6.14. SQL query: algebraic optimization + index nested join

- In this example we consider the SQL query and the same data
  - We also assume an index on  $s(B,D,E)$  by the join attribute  $B$
  - Recall:  $A > 11$  is satisfied by 40% tuples of  $r$
- Strategy
  - Do not preprocess  $r$  or  $s$ ; ignore the projection below  $r \bowtie s$  (shown in grey), but be mindful of on-the-fly application of selections to  $r$  and  $s$



- Use  $r$  as the outer relation, and only for each tuple satisfying  $r.A > 11$ , consider its  $B$ -value, consult the index on  $s$  and if a match is found further verify that  $s.E=55$  is satisfied.
- for every successful match deposit  $r.A, r.C, s.D$  to the output buffer
- Cost
  - Instream  $r$ : 200 page accesses, 1 buffer
  - Consult the index  $8,000 * 0.40 = 3200$  times
  - Cost of consultation if the index:
    - Case of B+tree:  $3200 * 3 = 9600$  page accesses; 1 buffer
    - Case of hashing:  $3200 * 1.25 = 4,000$  page accesses; 1 buffer
  - Total with B+tree:  $200 + 9600 + 3 = 9803$  page accesses; 3 buffers
  - Total with hash index:  $200 + 4000 + 3 = 4,203$  page accesses; 3 buffers

## 6.15. Query processing: summary

- Query processing does wonders for performance
- Optimization does not usually require user intervention
- An optimizer considers many strategies
  - A relation may be preprocessed writing a result temporarily on disk
  - Sorting as an option is considered
  - Index may be consulted dynamically
  - Selections and projection may be applied on-the-fly
  - Wise use of buffers
  - Statistics is taken into account in estimations
  - Several plans may be considered, most inexpensive one is executed
- What is the cost of determining the best plan?
  - Plans only require catalog, including the statistical information
  - A catalog may be cached in memory when loading a database
  - Assuming this consideration of different plans and determination of the most efficient one costs no page accesses
- Some reflections
  - Who produces the executable code? The system does
  - Imagine how difficult it would be if the user had to hand-write a Java program to do what an optimized plan does?
  - Moreover, the optimization strategy may change dramatically if there are changes in sizes of relations, new indexes are created or existing ones removed, and more statistics becomes available
  - This chapter shows the viability of the algebraic nature of SQL
  - A robust performance model should take cost of processing into account and take advantage of physical contiguity of pages on the disk





## Chapter 7

### *Database schema design with entity-relationship model*

- The entity-relationship (ER) model
  - Entities
  - Relationships
  - Inheritance
  - Participation of entities in relationships
- Translation of ER schema to relational database schema
  - Phase 1: mechanical translation
  - Phase 2: refinement
  - Phase 3: SQL schema
- Caution: our terminology and notations ...
  - The terminology and graphical notations vary from one publication (book or a paper) to another
  - We hope our terminology is clearer and notation intuitive

## 7.1. Database schema design

- The concepts of *schema* and *state* are important dual terms in databases
  - A schema represents structure – hypothesized to remain fixed
  - An instance or state consists of concrete data it can change with time but it must always conform to the schema
  - A schema is designed for a specific database application and then ...  
all subsequent development of a database application is only based on the schema and independent of instances
- To model an enterprise, a schema must be designed carefully
  - Multiple schemas for an enterprise are possible
  - Application development is dependent on a specific schema
  - Applications relative to one schema may not work on another schema
  - No automation exists for translating an application based on one schema to another
  - We should be prepared to be locked in to the schema we choose
  - This means a schema must be designed carefully
  - If possible, design should take some future evolution of data into account so that the schema does not have to be changed
- Tools for design of database schema
  - Use entity-relationship (ER) model: considered here
  - Use Dependency theory: will discuss briefly in this course
  - There are no perfect or fully automated tools for schema design

## 7.2. How ER sees the real world

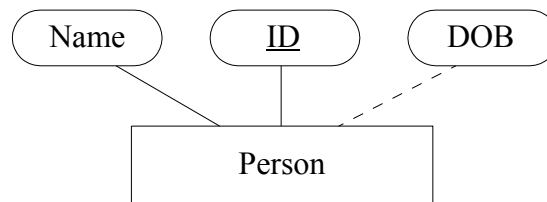
- In ER the real world consists of entities and relationships
  - An entity is a uniquely identifiable object in the real world
  - There are relationships among entity types
  - ER also recognizes hierarchy among entity types

## Entities, entity sets, and entity types

- The real world can be partitioned into entities that are similar
  - Entities having same types of properties are considered similar
  - Similar entities are considered to constitute an *entity-set*
  - All entities in an entity-set can be uniformly typed by an *entity type*  
An entity type consists of a name and attributes  
some attributes facilitate unique identity of entities
- The ER representation
  - The name of entity type is shown in a rectangle
  - Attributes are shown in ovals  
Ovals are connected to the rectangle by edges that are solid or broken
  - An attribute with broken edge may not be known for some entities

### 7.3. Example of entities, entity type, and entity set

- Consider Person entity type shown represented in ER style



- This says that a Person entity, as an instance of Person type, has a Name, ID, and DOB (date of birth)

The broken edge means we may not know the DOB of some Person entities

The ID attribute provides unique identity to every Person entity

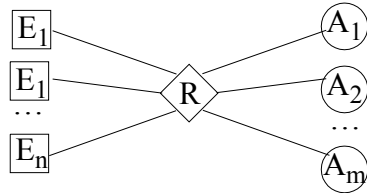
- An example of Person entity set as an instance of Person type

| Name | ID   | DOB      |
|------|------|----------|
| John | 2343 | 70/12/24 |
| Mary | 3126 | 72/03/30 |
| Hari | 2312 |          |

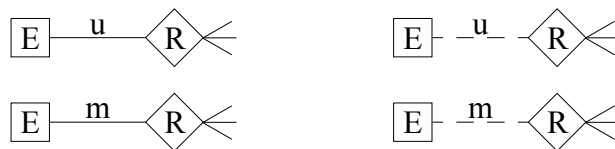
- This Person entity set consists of three Person entities
- Note that we have overloaded the term “Person”
  - It stands for entity type, entity instances, and entity set
  - The specific usage should be clear from the context
  - No confusion should arise

## 7.4. Relationships, relationship sets, and types

- Entities of certain types can have relationships
- The ER representation



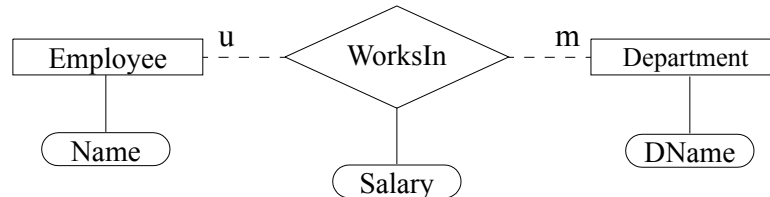
- The name of a relationship type is shown in a diamond (see R above)  
A relationship may have attributes of its own (see A<sub>1</sub>, ..., A<sub>m</sub> above)
- Relationship type It is connected to entity types (see E<sub>1</sub>, ..., E<sub>n</sub> above)
- Edge between entity and relationship types
  - Edges are solid or broken and labeled “u” or “m”
  - Consider a relationship type R and an entity type E



- Solid edge from E means that every entity of type E participates in some instance of R
- Broken edge from E means that some entities of type E may not participate in any instance of R
- The label “u” (uni) means that an entity of type E participates in at most one instance of R
- The label “m” (multi) means that an entity of type E may participate in multiple instances of R

## 7.5. Example of relationship instance

- Consider the following ER schema



- A relationship set as an instance of WorksIn relationship type

| Name |
|------|
| John |
| Mary |
| Hari |
| Leu  |

(a) Employee  
entity set

| DName |
|-------|
| Shoes |
| Toys  |
| Auto  |

(b) Department  
entity set

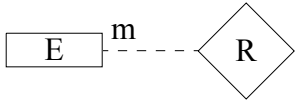
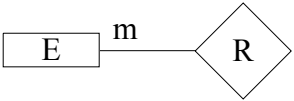
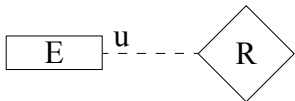
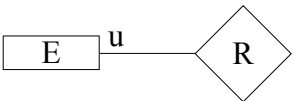
| Name | DName | Salary |
|------|-------|--------|
| John | Toys  | 50K    |
| Mary | Toys  | 60K    |
| Hari | Auto  | 55K    |

(c) WorksIn  
relationship set

- Note that there are 3 instances of WorksIn relationship
- Leu and Shoes do not participate in this relationship (see broken edges in the schema)
- Note that Name (the key of Employee entities) is a key of WorksIn relationship  
This is because of the label “u” between Employee and WorksIn
- DName is not a key of WorksIn relationship  
This is because of the label “m” between Department and WorksIn  
Not surprisingly Toys occurs twice in WorksIn relationship set
- In general every occurrence of the label “u” gives rise to a key that is independent of other keys
  - Thus multiple keys for a relationship type are possible

## 7.6. Keys of relationships

- We have seen an example where key of an entity type can serve as a key of relationship type
  - Here we distill what happens in general
- Consider a relationship type R
  - Several entity types can participate in R
  - Whether the key of a specific entity type can serve as the key of the relationship does not depend upon other relationships
- Therefore, we focus on participation of a single entity E and R
  - There are four types of labeled edges between E and R:

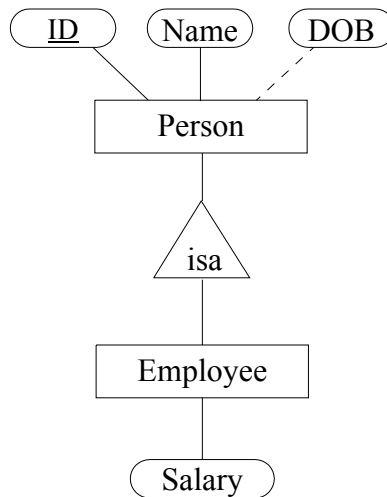
|             | Partial (Dotted)                                                                                                                                  | Total (Solid)                                                                                                                                |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Multi ("m") |  <p>Key(E) cannot be key of R<br/>Some E may not occur in R</p> |  <p>Key(E) cannot be key of R<br/>Every E occurs in R</p> |
| Uni ("u")   |  <p>Key(E) is a key of R<br/>Some E may not occur in R</p>     |  <p>Key(E) is a key of R<br/>Every E occurs in R</p>     |

- Every entity type connected with an edge labeled "u" contributes a key independent of others
  - Thus several keys for R are possible; such keys are logical keys
  - How these are used or exploited physically is a different matter



## 7.7. ISA property

- ISA allows an entity to inherit from another
  - (In databases, inheritance seems to predate object-oriented systems?)
- Example

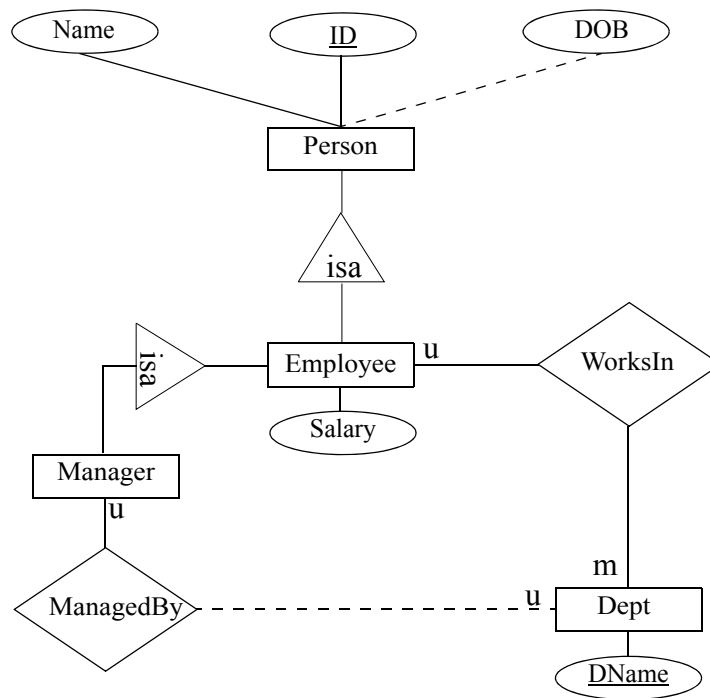


- Note that the isa triangle is directional
- This is read: employee isa person
- An Employee inherits ID, Name, and DOB attributes of a Person
- The key is also inherited
- In addition an Employee has Salary attribute

## 7.8. The role of Entity Relationship (ER) Model

- ER helps in understanding an application more thoroughly
- It is an excellent tool for designing a database schema
  - A schema in ER is in graphical format
- But ER is not a model for storing and processing data
  - Therefore an ER schema, once designed, has to be translated into a schema for a working database platform
  - Then an application can be developed on the chosen platform
  - We will consider the relational database platforms (such as Oracle, MySQL, DB2, etc.)
- We cover
  - Modeling of an enterprise as an ER schema
  - Mechanical translation of ER into a relational database schema
  - Intuitive hand improvement of relational database schema
  - Definition of the relational database schema in SQL

## 7.9. Personnel database: an example of ER Schema



- At type level we see ...
  - Four entity types: Person, Employee, Dept, and Manager
  - Two relationship types: WorksIn and ManagedBy
  - Two occurrences of isa: Employee isa Person and Manager isa Employee (who in turn isa person)
- In an instance
  - Every employee works in a unique department
  - Every department has at least one employee
  - If a department has a manager it is unique
  - Every manager manages a unique department

## 7.10. Translation from ER to relational database schema

- ER model only helps in designing a database
  - It does not have a storage model or a retrieval language
  - It must be translated to schema of a “real” database system
- We translate to relational schema; this is done in three phases
- Phase A. Mechanical translation from ER to relational
  - Each entity is translated to a relational schema
  - Through “isa” include key attributes by reference  
Designate a key, make a note of other possible keys
  - Each relationship is translated into a relation schema  
Draw keys of participating entities by reference  
Include local attributes of relationship
- Phase B. Refine the relational database schema
  - Remove redundancies
  - Not all entities participate in relationships. Therefore, we should be careful when merging relations having different origins
- Phase C. Give SQL schema using create table, constraints and triggers
- At every juncture in phases A, B, and C
  - Note down discrepancies, eliminating as many as possible
  - Informs the database owner of the remaining discrepancies

## 7.11. ER to relational translation for Personnel database

- Phase A: Mechanical translation
- For Personnel ER schema we obtain following relations
  - Person (Name, ID, DOB: nullable); note that ID is the key
  - Employee (ID → Person, Salary); “→” is read “references”  
Name and DOB are not included
  - Dept (Dname)
  - Manager (ID → Employee)
  - WorksIn (ID → Employee, Dname → Dept)  
Because Employee uni participates, ID alone is a key  
(In case of multi participation, ID+Dname would be the key.)  
*Residual Note 1: Every employee participates in WorksIn*
  - ManagedBy (Dname → Dept, ID → Manager)  
Manager as well as Dept uni participate  
Therefore, we could designate either ID or Dname as the key  
We have chosen to designate ID as the key  
*Residual Note 2. We can designate Dname as the key if we wish*  
*Residual Note 3. A manager must be attached to a department*
- Summary of Phase A
  - There are six relations in the relational database schema
  - All objects have their existence in relations arising from entities, but some of them may not show up in relations arising from relationships
  - In addition, there are three *residual notes* to be taken care of
  - We must keep all these in mind in Phase B

## 7.12. ER to relational translation for Personnel database

- Phase B: Refinements
- We inspect relations in Phase A and group overlapping ones
- Group 1
  - Person (Name, ID, DOB: nullable)
- Group 2
  - Employee (ID → Person, Salary)
  - WorksIn (ID → Person, Dname → Dept)
  - Residual Note 1: Every employee participates in WorksIn*
  - Note that every employee has a salary and a department
  - Collapse the two relations and Residual Note 1 into the following  
Emp (ID → Person, Dname → Dept, Salary)
- Group 3
  - Dept (Dname)
  - Manager (ID → Employee)
  - ManagedBy (ID → Employee, Dname → Dept)
  - Residual Note 2. We can designate Dname as the key if we wish*
  - Residual Note 3. A manager must be attached to a department*
  - Note that every department may not have a manager
  - Collapse the relations and residual notes into the following
  - Dept (Dname, ManagerID → Emp: nullable)
  - Residual Note. A manager manages a unique department*

## 7.13. ER to relational translation for Personnel database

- Phase C: SQL definition
- We obtained the following relational design
  - Person (Name, ID, DOB: nullable)  
Emp (ID → Person, Dname → Dept, Salary)  
Dept (Dname, ManagerID → Emp: nullable)  
*Residual Note. A manager manages a unique department*
  - Note a difference between how we and SQL treat nulls.  
In our case non-nulls are default and in SQL nulls are default
- SQL declaration

```
create table Person (
 ID char (9) not null,
 Name char (30) not null,
 DOB date,
 primary key (ID))

create table Emp (
 ID char (9) not null references Person,
 Dname char (12) not null references Dept,
 Salary integer not null,
 primary key (ID))

create table Dept (
 Dname char (9) not null,
 ManagerID references Emp,
 primary key (Dname),
 check ((select count (*)
 from Dept m
 where m.ManagerID <> null
 group by m.ManagerID) = 1)
```
- In this case no discrepancies are left





## Chapter 8

### *Hybrid languages: JDBC*

- Mixing SQL and general purpose programming languages
  - Why is it needed
  - The main advantages over plain SQL commands
- JDBC – Java based database connectivity
  - Syntax
  - Efficiency issues in hybrid programs

## 8.1. Java Database Connectivity (JDBC)

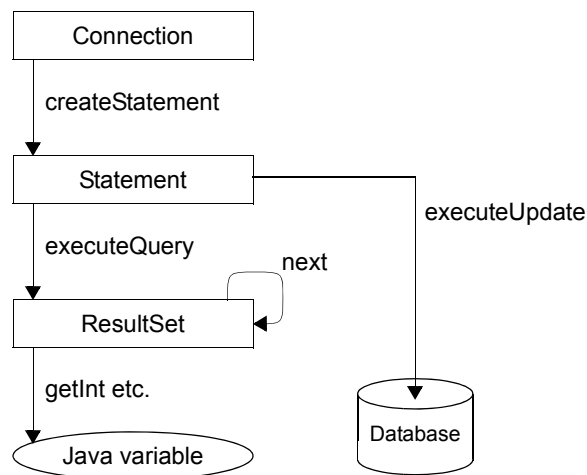
- JDBC is a hybrid of SQL and general purpose programming language Java
  - Java acts as a host language for issuing SQL commands
  - It is similar to ODBC, open database connectivity
  - There are other hybrid environments such as Oracles's SQL/PL
  - JDBC is object-oriented and it has grown more systematically
- In these slides we discuss
  - Hybrid between SQL and general purpose programming language
  - JDBC syntax in some detail
  - How to write efficient JDBC programs

## 8.2. JDBC = A driver + java.sql package

- One needs a driver to connect to a database
  - The drivers are specific to a database platform
  - Drivers are typically manufactured by database or third party vendors
  - In our demo we use a driver supplied by Sun that is bundled with JDK
- JDBC essentially consists of java.sql and javax.sql packages
- We mainly concentrate on java.sql package
- JDBC is huge, we concentrate on the core part

### 8.3. java.sql package

- java.sql consists of interfaces and classes
  - The diagram below shows classes and interfaces as rectangles
  - Directed arrows shows transition from one object type to another
  - Labels show functions



- Connection objects allow a connection through a driver
- Statement object can hold SQL statement to be executed
- An update statement can be executed to update the database
  - In “executeUpdate” the term “update” is drawn from English
  - It refers to execution of insert, delete and update of SQL
- A query is executed to retrieve a ResultSet object
  - A resultset can be scanned one tuple at a time
  - Attribute values can be brought into ordinary Java variables

## 8.4. ResultSet

- The concept of a ResultSet is the core of all hybrid languages
- A ResultSet is used as an iterator
  - A resultset is opened automatically when a query is executed:  
`rs1 = stmt1.executeQuery ("... an SQL query ...");`
  - `rs1.next()` returns a boolean; it returns TRUE if there is a next tuple  
the function `next()` also brings the next tuple in `rs1`
  - `rs1.close()` closes `rs1`
- A resultset allows tuple-at-a-time scan
- Contents of a tuple can be brought into Java variables
- Complex decisions can be made and actions can be taken
- This is what brings the power of a general purpose programming language at our disposal
- This is far more powerful than what can be done in SQL
- Another term for the resultset in hybrid languages is cursor

## 8.5. An example in 7 slides: Steps in a JDBC program

- Example: Consider Emp(Name, DName, Salary) relation
  - We want to create a report of employees in Toys department
  - We want to transfer two employees from Toys to other departments
- The JDBC program will consist of the following steps
  - Step 0. Import the java.sql package.
  - Step 1. Load and register the driver.
  - Step 2. Establish the connection to the database
  - Step 3. Create a statement that is a query on employees in Toys
  - Step 4. Execute the statement to create a resultset
  - Step 5. Process the resultset to print a report
  - Step 6. For the existing connection, create a PreparedStatement  
A prepared statement is compiled once executed many times
  - Step 7. Execute the prepared statement twice, once for each update
  - Step 8. Commit transactions on the connection
  - Step 9. Close the statement and connection
- We show these steps in detail in the following slides

## 8.6. An example in 7 slides: import, load and register

- Step 0. Import the java.sql package

```
// Step 0. Import Java.sql package
import java.sql.*;
```

- Step 1. Load and register a JDBC driver

```
// Step 1: Load the driver
try {
 // Load the driver (registers itself)
 Class.forName ("com.mysql.jdbc.Driver");
}
catch (Exception E) {
 System.err.println ("Unable to load driver.");
 E.printStackTrace ();
}
```

- Driver "sun.jdbc.odbc.JdbcOdbcDriver" is bundled with JDK
- The DriverRegister method does not appear in the code.
- The driver calls the DriverManager.registerDriver() to register itself
- There are four types of drivers, not considered here

## 8.7. An example in 7 slides: establishing a connection

- Step 2. Establishing a connection

```
try {
 // Step 2. Connect to the database
 Connection conn1; // An object of type connection
 String dbUrl = "jdbc:odbc:DemoDataSource";
 String user = "";
 String password = "";
 conn1 = DriverManager.getConnection (dbUrl, user, password);
 System.out.println ("*** Connected to the database ***");
 . . .
} // End of try
catch (SQLException E) {
 System.out.println ("SQLException: " + E.getMessage());
 System.out.println ("SQLState: " + E.getSQLState());
 System.out.println ("VendorError: " + E.getErrorCode());
} // End of catch
```

- The code for connection embedded in a try-and-catch page  
This allows exceptions to be handled by exception handler
- Alternate syntax allows prompting for username and password
- Connections are complicated and fragile objects  
A connection may not succeed in the first attempt  
A connection may get broken unexpectedly
- Therefore, parameters such as “MaxReconnects” are allowed  
Google “MaxReconnects” to learn more
- Parameters have reasonable default values associated with them
- For noisy connections higher MaxReconnects may be provided
- These details are very interesting, but not considered in the course
- From a JDBC program, connections can be made to multiple data-bases, spreadsheets, text files, even residing on diverse platforms



## 8.8. An example in 7 slides: create and execute statement

- Step 3. Create a Statement

- A statement is a container for an SQL command
- It has to be associated with a connection

```
// Step 3A. Create stmt1 object and associate it with conn1
Statement stmt1 = conn1.createStatement ();
```

- Step 4. Execute the Statement

```
// Step 4A. Execute a query, receive result in a result set
ResultSet rs1 = stmt1.executeQuery ("select e.Name, e.Salary" + " " +
 "from Emp e" + " " +
 "where e.DName = 'Toys' ");
```

- Note a blank (" " +) is forced between two lines
- The resultset object is created to hold the result of the query

## 8.9. An example in 7 slides: process result set

- Step 5. Process the result set

- The concept of a resultset has already been discussed

// Step 5A. Process the result set

```
...
int count = 0;
double totalPayroll = 0.0;
String jName; // To store value of Name attribute
double jSalary; // To store value of Salary attribute
while(rs1.next()) {
 // Access and print contents of one tuple
 jName = rs1.getString(1); // Value accessed by its position
 jSalary = rs1.getInt("Salary"); // Access by attribute Name
 System.out.println(jName + " " + jSalary);
 count = count + 1;
 totalPayroll = totalPayroll + jSalary;
}
// Clean up
rs1.close();
stmt1.close();
...
```

- get functions are used to extract attribute values in java variables  
For an integer attribute getInt( ) is used, similar for other types
- Attribute values can be referred to by their position or name  
For example, see rs1.getString(1) and rs1.getInt("Salary")
- Attribute values are deposited in variables of the host language (Java) in order to make decisions based upon their values.
- In the sample code jSalary and jName are such variables.
- Just like an ordinary Java program, a count and running total of salaries are calculated in the loop.
- Contents of the tuple are also printed in the loop
- On exit from the loop the count and grand total of salaries are printed

## 8.10. Prepare and execute updates

- We use a `PreparedStatement`
  - Such statement is compiled once, executed many times
  - Leave provision for simple variable substitution

// Step 6. Create a Prepared Statement object, reuse connection Conn1  
`PreparedStatement stmt2 =`

```
 conn1.prepareStatement ("update Emp" + " " +
 "set DName=? , Salary=?" + " " +
 "where Name = ? ");
```

// Execute stmt2: Move Hari to Shoes with a salary of \$65,000:

```
stmt2.setString(1,"Shoes");
stmt2.setInt(2,65000);
stmt2.setString(3,"Hari");
stmt2.executeUpdate();
```

// Execute stmt2: Move Leu to Credit with a salary of \$75,000

```
stmt2.setString(1,"Credit");
stmt2.setDouble(2,75000.0);
stmt2.setString(3,"Leu");
stmt2.executeUpdate();
```

// Cleanup

```
stmt2.close();
```

## 8.11. Finishing up: closing and committing

- A JDBC program may make many queries and updates
  - Queries do not change a database
  - But updates do
  - What if the system crashes in the middle of execution?
  - We may not know what updates have been completed
  - It would be best if could think of updates as a batch either *all* updates in a given batch are made or *no* update is made
  - The commit function on a connection accomplishes this
  - It should be the last function to be executed on a batch of updates
- This leads to the following code in our example

```
// Step 8. Commit transactions on conn1 and close it
conn1.commit();
```

- Cleanup

```
// Step 9. Close statements and connections
conn1.close();
```

## 8.12. Efficiency and Logic in JDBC (3 slides)

- The efficiency in SQL is usually deferred to the system
  - Database do a superb job at optimizing SQL queries
  - But general purpose languages such as Java are poor in this respect
  - Often JDBC programs are written carelessly; they deprive the database system of opportunities to do optimization.
  - This situation arises especially when simultaneous traversal of two or more relations is needed.
- Example: Consider two relations  $r(A,B,C)$  and  $s(B,D,E)$ 
  - Relations  $r$  and  $s$  occupy 200 and 400 pages, respectively
  - The relation  $r$  has 8,000 tuples. Of these, 40% satisfy  $A > 11$ .
  - The number of tuples in  $s$  do not matter in this example
  - Consider the following SQL-like query.

```
select x.A, x.C, y.D
from r x, s y
where x.B = y.B
 and x.A > 11
 and y.E = 55
 and f(A,B,C,D)
```
  - Here,  $f$  is a function that requires Java.

### 8.13. Efficiency and Logic in JDBC: Slide 2 of 3

- The following two JDBC solutions are possible
- First solution
  - Execute the following query

```
select x.A, x.B, x.C, y.D
from r x, s y
where x.B = y.B
 and x.A > 11
 and y.E = 55
```
  - This query is same as in the chapter on optimization except x.B is now included in the output
  - This changes the size of output:  
Previously:  $\lceil 8000 * 0.40 * 0.04 * 20 / 1000 \rceil = 3$  pages  
Now:  $\lceil 8000 * 0.40 * 0.04 * (20 + 8) / 1000 \rceil = 4$  pages
  - The total cost becomes 603 page accesses instead of 602
  - In a single loop execute  $f(A,B,C,D)$
- Second solution:
  - Execute the following query and scan r-tuples in a while loop in JDBC

```
select *
from r x
where x.A > 11
```
  - For each tuple in the result set of the above query, execute

```
select y.B, y.D
from s y
where y.E = 55
```
  - In a nested loop verify  $x.B = y.B$  and execute  $f(A,B,C,D)$

## 8.14. Efficiency and Logic in JDBC: Slide 3 of 3

- Cost of first solution
  - As shown in lecture on optimization, under reasonable circumstances this requires 603 page accesses needing 6 seconds
- Cost of second solution
  - Scan of r will require 200 page accesses. Let's assume that 40% of the tuples satisfy  $A > 11$ . Thus out of 8,000 tuples, 3,200 tuples will survive. Therefore, relation s will be scanned 3,200 times. Thus, the cost is  $200 + 3200 * 400 = 1,280,200$ . With page accesses at 10 ms per page this adds up to about 3.6 hours.
  - One may argue that since no state-dependent variable is passed from outer query to the inner query, the compiler should be able to determine that the inner result set need to be computed only once. Let's suppose the inner resultset consists of 13 pages. Therefore, scanning the inner relation will require  $3200 * 13$  page accesses, needing about 7 minutes.
- Significant gain in performance can be achieved by letting SQL rather than Java handle the nested loop
- One may have to sacrifice a slight ease in programming
  - The solution in the second style may be easier to write
  - Writing in the style of first solution may require more thought

The result set of a join interleaves two types of objects

This interleaving may be too haphazard for stream-based processing

By requiring the SQL query to sort the resultset, the result may become more conducive stream-based processing in a single pass

## 8.15. When not to use Java arrays

- The concept of streaming data is fundamental in databases
  - By default such streams can be very large
  - Streams should be handled as a stream that is passing by
  - Temptations to store streaming data in arrays must be avoided
  - Senseless if the array will be processed in the same sequence as the original stream
- An instructive example is calculation of average salary
  - We only need to keep track of the running total and count
  - No need to store employee tuples in an array  
This will avoid wasting memory and slowing the application
  - The logic of an array-based solution can be applied directly to the streaming data
  - The reader should consider streaming of pairs of objects and consider carefully how to handle such a stream without wasting resources.
- A word of advice ...
  - Our projects use small databases to keep them manageable for you
  - But your code *must* reflect the mind-set that the databases are large
- When is an array appropriate?
  - Must give serious thought before using it.



## 8.16. Further remarks

- Alternative logic surrounding a result set
  - In DemoJDBC.java, the next() function is used as a certification for entering the body of a while loop.
  - An alternate logic is to execute next() once before entering the loop and once at the end of the body of the while loop:

```
rs1.next();
while (not eof) {
 do something with the tuple
 rs1.next()
}
```
- More recent versions of JDBC allow result sets to be traversed forward and backward and certain jumps
- JDBC allow query of metadata
  - One can connect to a database one knows nothing about
  - Copy the contents of the entire database!



## Chapter 9

### *Object-orientation in databases*

- SQL is algebraic at its core
  - The algebraic constitution is the source of its strength and weakness  
It is fruitful to think of it as the mother of query languages
  - We have seen that SQL works well for atomic data
  - The big question is ...  
... how to extend the “SQL” for more complex forms of data
- In this chapter we add object-orientation to SQL
  - This leads to OQL – the Object Query Language
  - Our treatment is informal and not to be equated to the “standard” OQL
  - Our main focus is on how one improves linguistic interface for users

## 9.1. General purpose languages vs. query languages

- Database query languages
  - Query languages are about and only about collections of objects
  - Queries draw subsets of collections and do not compute new info
  - The ability to query is at the core of databases ...
  - A query language should be easy to use and amenable to optimization
    - something that general-purpose programming languages, such as Java, *cannot* deliver

Because of its algebraic constitution SQL does deliver this ...

... but SQL covers only atomic data that has no structure of its own
- Difficulty in going from simple atomic values to objects
  - Unlike general purpose languages such as Java, algebraic query languages are too weak to allow construction of arbitrary objects
  - For advancing database frontier we should *try* collections with *predefined object-structures* and SQL style select–from–where
- In this book we consider three interesting special cases
  - In this chapter: values with object-orientation – an intro to OQL
  - In Chapters 11 and 12: semistructured xmlized values, and XQuery
  - In Chapter 13: Combine the above to obtain OOXML and OOXQuery
- An introduction to OQL ...
  - ... as stated before it is the main objective of this chapter
  - We will consider simple objects with values that are atomic, nested relations, references, and support inheritance

## 9.2. Nested values

- In the classical relational model ...
  - An (attribute) value is atomic
  - The main vehicle for navigation is A op B (e.g. e.DName = d.DName)
- A nested value is a relation
- Example: consider the following schema for Emp relation

Emp

| Name | DName | Salary | Child |     |
|------|-------|--------|-------|-----|
|      |       |        | CName | Age |

- An employee has a name, department, and a salary
  - The employee also has children, listed under the Child attribute
  - The value of the Child attribute is a relation
  - Each tuple of Child-relation gives the name and age of a child
- An instance is a nested relation:

Emp

| Name    | DName | Salary | Child |     |
|---------|-------|--------|-------|-----|
|         |       |        | CName | Age |
| John    | Toys  | 50K    | Jack  | 18  |
|         |       |        | Jill  | 15  |
|         |       |        | Hill  | 10  |
| Twinkle | Shoes | 45K    | Star  | 17  |

- This scheme can be denoted as follows
  - Emp(Name, DName, Child(CName, Age))

### 9.3. Querying nested-relations

- We discuss some queries informally
- Example: List Names of employees in Toys department and info about their teenage children

```
select e.Name, (select *
 from e.Child c
 where 12 < c.Age and c.Age < 20)
from Emp e
where e.DName = 'Toys'
```

| Name | Child |     |
|------|-------|-----|
|      | CName | Age |
| John | Jack  | 18  |
|      | Jill  | 15  |

- Example: What does the following query return?

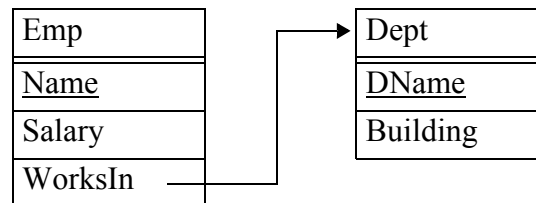
```
select e.Name, c.CName, c.Age
from Emp e, e.Child c
where e.DName = 'Toys' and (12 < c.Age and c.Age < 20)
```

| Name | CName | Age |
|------|-------|-----|
| John | Jack  | 18  |
| John | Jill  | 15  |

- How would we define union, difference, selection, projection, join, and possibly some new operators on nested relations?
  - About a dozen interesting papers on this in the database literature

## 9.4. Reference values

- Relational model fragments related info across relations
  - Natural join is needed to connect one to another
  - $e.DName = d.DName$  takes us from Emp tuple to Dept tuple
- How to make “d.DName” a department?
- Example: Consider the following schema



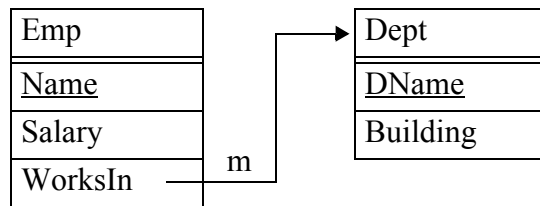
- Here, Name, Salary, DName, and Building are ordinary values
- WorksIn is a pointer-value that points to a specific department  
Textually we may use the phrase “WorksIn: references Dept”
- If  $e$  is a variable that ranges over Emp ...
  - Then  $e.Name$  is name of an employee, say John
  - $e.WorksIn$  is a department object, a tuple-object
  - $e.WorksIn.Building$  gives the building where John works
- List names and buildings of employees earning at least 100K

```
select e.Name, e.WorksIn.Building
from Emp e
where e.Salary >= 100000
```

  - The query does not need two relations in the from clause
  - No natural join ( $e.DName = d.DName$ ) is needed at user level

## 9.5. Nested References

- What if an employee worked in several departments?
  - The edge from WorksIn attribute to Dept type can be labeled “{”
  - Textually we may use the phrase “WorksIn: references {Dept}”
  - Graphically we use “m”



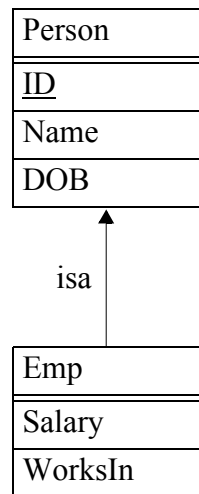
- Reconsider: List names and buildings of employees earning at least 100K

```
select e.Name, (select d.Building
 from e.WorksIn d)
from Emp e
where e.Salary >= 100000
```
- The phrase “e.Name” of previous example does not change  
But “e.WorksIn.Building” has become a select statement



## 9.6. Inheritance

- How can inheritance help?
  - Suppose we wish to say that an “employee is a person”

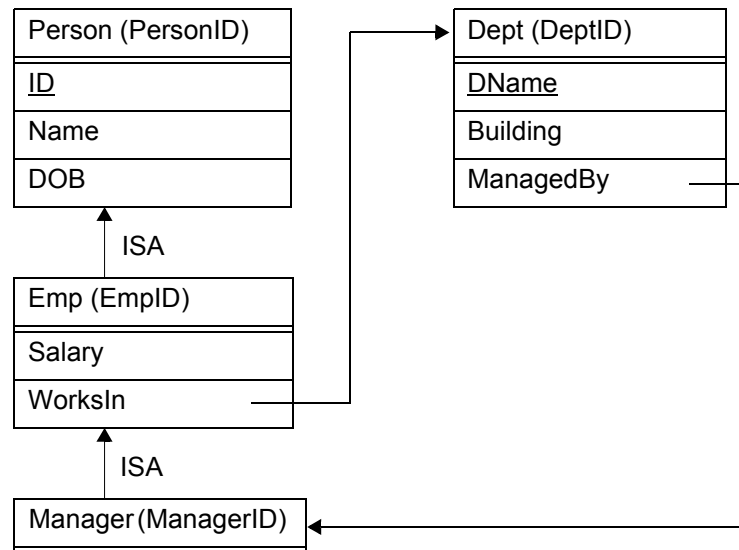


- Note that Name has migrated to Person
- Then if  $e$  ranges over employees, then  $e.DOB$  should make sense
- Example. List Name, DOB and Salary of all employees  

```
select e.Name, e.DOB, e.Salary
from Emp e
```
- An object-oriented model would support this linguistics

## 9.7. Personnel database: an object-oriented scheme

- Put references and inheritance together



- What is date of birth of John's manager?

```
select e.WorksIn.ManagedBy.DOB
from Emp e
where e.Name = 'John'
```

- Phrase `e.WorksIn.ManagedBy.DOB` is friendly
  - `e` is John's tuple; `e.WorksIn` is John's department
  - `e.WorksIn.ManagedBy` is John's manager
  - The manager is an employee, hence also a person  
Therefore, `e.WorksIn.ManagedBy.DOB` is John's manager's DOB

## 9.8. Comparison with classical SQL

- Recall the following (for *date of birth of John's manager*)

```
select e.WorksIn.ManagedBy.DOB
from Emp e
where e.Name = "John"
```

- How will we do it in classical relational database setting?
- A relational design for Personnel is as follows

```
Person (Name, ID, DOB)
Emp (EmpID → Person.ID, DName → Dept, Salary)
Dept (DName, ManagerID → Emp)
```

- We assume that there are no nulls

- In SQL “*date of birth of John's manager*” is as follows

```
select p2.DOB
from Emp e1, Person p1, Dept d, Emp e2, Person p2
where p1.Name = "John"
 and e1.EmpID = p1.ID
 and e1.DName = d.DName
 and d.ManagerID = e2.EmpID
 and e2.EmpID = p2.ID
```

- A comparison of the two queries
  - A single variable *e* vs. five variables *e1*, *p1*, *d*, *e2*, and *p2*
  - *e.Name* = "John" vs. *e1.EmpID* = *p1.ID* and *p1.Name* = "John"
  - *e.WorksIn* vs. *e1.DName* = *d.DName*
  - *.ManagedBy* vs. *d.ManagerID* = *e2.EmpID* and *e2.EmpID* = *p2.ID*
- Clearly ...
  - ... OQL is linguistically far more natural for users than plain SQL
  - But let us not forget that OQL is “derived” from SQL

## 9.9. Object-oriented databases

- Object-oriented db issues are more complex than relational db
  - An agreement on a standard is elusive
  - Implementation is expensive for industry
  - Legacy SQL users may be ambivalent about switching to OQL
- ODMG (Object data management group) gave a standard
  - Addressed many important issues in object-oriented databases
  - Claims to allow arbitrary objects
  - Allows a large variety of collections:  
sets (relations), bags, arrays, tables, etc.
  - Two different levels of inheritance
  - Supports minimal level of multiple inheritance
  - Encapsulation of methods and properties
  - Properties are: attributes and binary relationships (as in ER-Model)  
(The references in our presentation do not fully cover relationships)
  - Unfortunately, ODMG did not succeed:  
OQL, the query language of ODMG is difficult to find
  - Lack of initiative and legacy issues seem to have arrested progress
- XML
  - XML is a powerful language for modeling data  
XQuery is its query language – also derived from SQL
  - In the next chapter we will give our own object orientation to XML  
and XQuery

## Chapter 10

### *Data dependencies and database schema design*

- We have covered database schema design using entity relationship approach in Chapter 7
- In this chapter we cover database schema design subject to constraints called data dependencies
- How is an application presented to us and what we want?
  - We are given the following about a database application:
    - Attributes (column names) in the database
    - Constraints that are required to be satisfied by all states of the database
  - Our objective is to propose a good database schema for the application
- About constraints, legal states, and updates
  - The constraints to be considered in this chapter are grounded in attributes (column names) in the database – further details will follow
  - A database state is said to be *legal* if it satisfies the constraints
- To be a good design, the proposed schema should support ...
  - Non-redundancy – so that the same information does not have to be stored in multiple places
  - Losslessness – to ensure that the information is not corrupted
  - Constraint preservation. When relations are updated the dependencies must be enforced to ensure that the database always stays in legal state

## 10.1. The challenge

- A given design can lead to redundancy in some relations
  - The removal of redundancy from a relation may require the relation to be broken (decomposed) into two or more relations

Example. In order to remove redundancy a relation  $r(ABC)$  may have to be broken into relations  $r_1(AB)$  and  $r_2(BC)$

- But, this may disrupt losslessness and preservation of constraints

Losslessness. The collective information in  $r_1$  and  $r_2$  should be the same as that in  $r$  – we must have  $r_1 \bowtie r_2 = r$  – but this may fail

Constraint preservation. Relation  $r_1$  “may not be aware of”  $C$  and  $r_2$  “may not be aware of”  $A$ . Thus a constraint involving  $A$ ,  $B$  and  $C$  can be enforced when  $r$  is updated, we may not have a strategy to enforce it when relation  $r_1$  or  $r_2$  are updated in isolation of each other

- The challenge is in choosing “good” decompositions ...
  - ... that help us ensure non-redundancy, losslessness, and preservation of constraints
- A sneak peek into non-redundancy
  - We will first introduce a notion of non-redundancy called BCNF (Boyce Codd Normal Form)
  - We will see that we can only guaranty a design where BCNF and losslessness are supported – but dependency preservation may fail
  - We will then introduce a less stringent notion of non-redundancy, called Third Normal Form, which can always coexist with losslessness and constraint preservation

## 10.2. Notation

- For convenience we introduce notation and conventions
  - Keep in mind that motivation for some notations is rooted in the fact that attributes are column names in relations
  - We will use  $A, B, C, \dots$  to denote attributes
  - We will use  $X, Y, Z, U, \dots$  to denote sets of attributes
  - $\{A\}$ , a set containing a single attribute  $A$ , may be denoted as  $A$
  - $X \cup Y$  may be denoted as  $XY$   
(here it helps to recall that  $X$  and  $Y$  are sets of column names)
  - $\{A, B, C\}$  may be denoted as  $ABC$
  - In specific case such as  $\{\text{Name}, \text{DName}, \text{Salary}\}$ , we may denote it as  $\text{Name DName Salary}$

### 10.3. Functional dependencies

- Syntax
  - Suppose  $R$  is a set of attributes and  $X$  and  $Y$  are subsets of  $R$ . Then  $X \rightarrow Y$  is a functional dependency over  $R$
- Example
  - $D \rightarrow B$  is a functional dependency over  $ABCD$ , but not over  $ABCE$
- Satisfaction of a functional dependency as a constraint
  - Suppose  $X \rightarrow Y$  is a functional dependency over  $R$  and  $r$  is a relation over  $R$ . We say that  $X \rightarrow Y$  is *satisfied* in  $r$  if whenever any tuples  $t_1$  and  $t_2$  of  $r$  agree on  $X$  then  $t_1$  and  $t_2$  also agree on  $Y$ .
  - If  $X \rightarrow Y$  is satisfied by  $r$ , we also say that  $r$  *satisfies*  $X \rightarrow Y$



## 10.4. Some examples of dependency satisfaction

- As an example, consider the relation  $r$  over ABCD

| A  | B  | C  | D  |
|----|----|----|----|
| a1 | b1 | c1 | d1 |
| a1 | b1 | c1 | d2 |
| a2 | b2 | c2 | d3 |

- The relation  $r$  satisfies  $AB \rightarrow C$ ,  $BC \rightarrow A$ , and  $D \rightarrow ABC$ ; but  $r$  does not satisfy  $C \rightarrow D$
- Of these, all are straightforward – except that sometimes  $D \rightarrow ABC$  seems to cause some confusion and we consider it in detail
- Proof of “ $D \rightarrow ABC$  is satisfied in  $r$ ”

We need to show that the implication “ $t_1$  and  $t_2$  agree on  $D \Rightarrow t_1$  and  $t_2$  agree on  $ABC$ ” is True for every pair of tuples  $t_1$  and  $t_2$  in  $r$

Let’s consider the Truth Table for implication

| P     | Q     | $P \Rightarrow Q$ |
|-------|-------|-------------------|
| True  | True  | True              |
| True  | False | False             |
| False | True  | True              |
| False | False | True              |

It is seen that for any tuples  $t_1$  and  $t_2$  in  $r$  it is never the case that “ $t_1$  and  $t_2$  agree on  $D$ ” is True but “ $t_1$  and  $t_2$  do not agree on  $ABC$ ” is False

Thus a scenario for the second row of the Truth Table never arises

Hence  $D \rightarrow ABC$  is satisfied in  $r$

## 10.5. Motivating example for redundancy

- Consider the Suppliers-Parts application described below
  - Attributes: Supplier, Address, Item, Price, denoted S, A, I, P
  - Constraints are the following functional dependencies
    - $S \rightarrow A$ , which says “a supplier has a unique address”
    - $SI \rightarrow P$ , which says “a given supplier can have only one price for a given item”
  - For convenience, suppose a relation over SAIP is named saip
- Consider the following state of saip relation

saip

| S    | A    | I    | P    |
|------|------|------|------|
| ABC  | Ames | Nut  | 0.50 |
| ABC  | Ames | Bolt | 0.60 |
| Acme | DSM  | Nut  | 1.00 |

- This state is legal as it satisfies the constraints  $S \rightarrow A$  and  $SI \rightarrow P$
- We see that  $S \rightarrow A$  leads to redundancy
  - In every tuple with a given supplier (an S-value), the same address (an A-value) is repeated
- The redundancy due to  $S \rightarrow A$  leads to following anomalies
  - Insertion anomaly. The address of a supplier has to be included every time an item is added
  - Modification anomaly. If a supplier address changes, it has to be corrected in multiple places
  - Deletion anomaly. If the relation has only one item left for a given supplier and this is deleted, then the supplier’s name and address are lost

## 10.6. Removal of redundancy

- Let's reconsider the saip relation

saip

| S    | A    | I    | P    |
|------|------|------|------|
| ABC  | Ames | Nut  | 0.50 |
| ABC  | Ames | Bolt | 0.60 |
| Acme | DSM  | Nut  | 1.00 |

- Removal of redundancy due to  $S \rightarrow A$ 
  - We use our intuition and decompose saip into sa(SA) and sip(SIP)
  - The contents of the database under new design are shown below

sa

| S    | A    |
|------|------|
| ABC  | Ames |
| Acme | DSM  |

sip

| S    | I    | P    |
|------|------|------|
| ABC  | Nut  | 0.50 |
| ABC  | Bolt | 0.60 |
| Acme | Nut  | 1.00 |

- Is the resulting design consisting of SA and SIP is good ...
  - Is the redundancy gone in the new design? Yes!
  - Is the new design lossless? Yes:  $\text{saip} = \text{sa} \bowtie \text{sip}$
  - Can  $S \rightarrow A$  and  $S \rightarrow I$  be preserved in the new design?  
Yes: Enforce  $S \rightarrow A$  via sa (when sa is updated) and  $SI \rightarrow P$  via sip (when sip is updated)
- What happens in general, how, and why?
  - We answer all these questions in terms of functional dependencies
  - Functional dependencies can be mysterious and require deeper understanding

## 10.7. Example of a lossy decomposition

- Not all decompositions are non-lossy
  - Consider the decomposition of saip into si and aip

| si   |      | aip  |      |      |
|------|------|------|------|------|
| S    | I    | A    | I    | P    |
| ABC  | Nut  | Ames | Nut  | 0.50 |
| ABC  | Bolt | Ames | Bolt | 0.60 |
| Acme | Nut  | DSM  | Nut  | 1.00 |

- We recompute saip as  $\Pi_{SI}(saip) \bowtie \Pi_{AIP}(saip)$ :

| saip |      |      |      |
|------|------|------|------|
| S    | A    | I    | P    |
| ABC  | Ames | Nut  | 0.50 |
| ABC  | DSM  | Nut  | 1.00 |
| ABC  | Ames | Bolt | 0.60 |
| Acme | Ames | Nut  | 0.50 |
| Acme | DSM  | Nut  | 1.00 |

- Obviously this is not the same as saip
 

ABC is now listed as being in Ames, as well as DSM

This is loss of information (even though number of tuples have increased)
- What is the reason for this loss of information?
  - $saip \neq \Pi_{SI}(saip) \bowtie \Pi_{AIP}(saip)$

## 10.8. Some observations on functional dependencies

- Functional dependencies generalize concept of a key
  - For example, saying “SI is a key in SAIP” can be expressed in terms of functional dependency  $SI \rightarrow SAIP$
  - But  $S \rightarrow A$  cannot be expressed as a key constraint on saip
- Some functional dependencies imply others
  - For example, if  $A \rightarrow B$  and  $B \rightarrow C$  are satisfied in a relation, then  $A \rightarrow C$  will also be satisfied in that relation
- Trivial functional dependencies
  - $X \rightarrow Y$  is said to be trivial if  $X$  is a superset of  $Y$
  - For example  $AB \rightarrow A$  and  $A \rightarrow A$  are trivial
  - Trivial functional dependencies are always satisfied and give us no information

## 10.9. Associating FDs with schemas

- FDs are a form of constraints in a database
  - They are meant to refer to relation schemas rather than relation states
  - The concept of dependency satisfaction is only a stepping stone to the concept of “holds” that is applied to relation schema
- Definition of “holds”
  - Suppose  $r$  is a relation over  $R$ , then we say that a functional dependency *holds* in  $R$  if it is satisfied in all states of the relations  $r$

## 10.10. Reasoning with functional dependencies: Implication

- *Implication* among functional dependencies

- Suppose the context: A set of functional dependencies  $\mathcal{F}$  and a functional dependency  $X \rightarrow Y$ .

We say that  $\mathcal{F}$  *implies*  $X \rightarrow Y$  if for all relations  $r$  that satisfy functional dependencies in  $\mathcal{F}$  also satisfy  $X \rightarrow Y$ .

- Example.  $\{A \rightarrow B, B \rightarrow C\}$  implies  $A \rightarrow C$

Proof. Suppose a  $r$  is a relation that satisfies  $A \rightarrow B$  and  $B \rightarrow C$ . Then we need to prove the  $r$  satisfies  $A \rightarrow C$ .

Suppose  $t_1$  and  $t_2$  are tuples in  $r$  that agree on  $A$ , i.e.,  $t_1(A) = t_2(A)$

Then because  $r$  satisfies  $A \rightarrow B$ ,  $t_1$  and  $t_2$  agree on  $B$

But then because  $r$  satisfies  $B \rightarrow C$ ,  $t_1$  and  $t_2$  agree on  $C$

Thus whenever two tuples in  $r$  agree on  $A$ , they also agree on  $C$

This means  $r$  satisfies  $A \rightarrow C$

- Problem with the proofs of implication ...

- These proofs invoke infinitely many relations that satisfy certain FDs
- One feels intuitively that the use of relations is only coincidental
- We like to have reasoning where only attributes need to participate
- This requires that we take certain “rules” for granted and apply them in deductions
- As an example if we assume the rule that allows us to deduce  $A \rightarrow C$  from  $\{A \rightarrow B, B \rightarrow C\}$ , then by double application of this rule we can deduce  $A \rightarrow D$  from  $\{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$

## 10.11. Deductions among functional dependencies

- Maier's rules of deduction

The following rules of deduction have been given by David Maier. Here,  $R$  is a relation schema and  $X$ ,  $Y$ ,  $Z$ , and  $W$  as subsets of  $R$ .

- Reflexivity.  $X \rightarrow X$  holds in  $R$ .
  - Accumulation. If  $X \rightarrow YZ$  and  $Z \rightarrow W$  hold in  $R$ , then  $X \rightarrow YZW$  also holds in  $R$ .
  - Projectivity. If  $X \rightarrow YZ$  holds in  $R$ , then  $X \rightarrow Y$  also holds in  $R$ .
- (Accumulation, adapted here, is stated a little differently by Maier.)

- What is interesting about Maier's rules?

- Soundness (correctness) of Maier's rules. If  $X \rightarrow Y$  can be deduced from  $\mathcal{F}$  using Maier's rules then  $\mathcal{F}$  implies  $X \rightarrow Y$

We will prove soundness shortly.

- Completeness of Maier's rules. If  $\mathcal{F}$  implies  $X \rightarrow Y$  then  $X \rightarrow Y$  can be deduced from  $\mathcal{F}$  using Maier's rules.

Completeness means that the three rules given by Maier are enough, no additional rules are needed for deductions

We have stated completeness without proof

- The deductions involve only attributes as symbols ...

... they do not invoke "states of relations" where the dependencies are satisfied, which – as remarked previously – can be troublesome as there are potentially infinitely many states



## 10.12. Soundness of Maier's rules

- A schema  $R$  and its subsets  $X, Y, Z, W$  are given.

- Assume  $r$  is an arbitrary relation over  $R$

- Reflexivity.  $X \rightarrow X$  (always) holds in  $R$  for any subset  $X$  of  $R$ .

Proof. This is obvious. It says that if two tuples in  $r$  agree on  $X$  then they agree on  $X$ . This means  $X \rightarrow X$  is satisfied in  $r$ . But as  $r$  is an arbitrary relation on  $R$ , this means  $X \rightarrow X$  holds in  $R$ .

- Accumulation. If  $X \rightarrow YZ$  and  $Z \rightarrow W$  hold in  $R$ , then  $X \rightarrow YZW$  also holds in  $R$ .

Proof. Assume that  $X \rightarrow YZ$  and  $Z \rightarrow W$  hold in  $R$  and tuples  $t_1$  and  $t_2$  in relation  $r$  agree on  $X$  (meaning  $t_1[X] = t_2[X]$ , i.e., the two tuples  $t_1$  and  $t_2$  have the same values on every attributes of  $X$ ); we must prove that the two tuples also agree on  $YZW$ .

As  $X \rightarrow YZ$  holds in  $R$ , it is satisfied in  $r$ . This means two tuples  $t_1$  and  $t_2$  agree on  $YZ$ . Hence they also agree on  $Z$ . Similarly, because of  $Z \rightarrow W$ , they agree on  $W$ . Obviously we can conclude that they agree on all of  $YZW$ . This means  $X \rightarrow YZW$  is satisfied in  $r$ . But since  $r$  is an arbitrary relation on  $R$ , this means  $X \rightarrow YZW$  holds in  $R$ .

- Projectivity. If  $X \rightarrow YZ$  holds in  $R$ , then  $X \rightarrow Y$  also holds in  $R$ .

Proof. Left as an exercise.

### 10.13. Monotonic deductions on functional dependencies

- Suppose we need to deduce  $X \rightarrow Y$  from a given set of functional dependencies  $F$ .

Then a style of deduction described below, called *monotonic deduction*, will always work

- Start with the instance of reflexivity  $X \rightarrow X$ .
- Apply accumulation until the right hand side is a superset of  $Y$ .
- Finally if necessary, use projection to obtain  $X \rightarrow Y$ .

- Example. From  $A \rightarrow BC$  and  $B \rightarrow DE$ , deduce  $A \rightarrow CE$

- Proof. Here is a monotonic deduction

We start with  $A \rightarrow A$  (reflexivity)

$A \rightarrow BC$  (given) leads to  $A \rightarrow BC$  (accumulation)

Because of  $B \rightarrow DE$ , we have  $A \rightarrow BCDE$  (accumulation)

From  $A \rightarrow BCDE$  deduce  $A \rightarrow CE$  (projection)

- Some observations

- Note that in this deduction we did not invoke any relation and satisfaction of dependencies in those relations

As stated before, there could be infinitely many relations

- The deduction, entirely based upon symbols, is finite
- A sound and complete set of rules of deduction were originally given by Armstrong. (See Appendix E)

Maier's rules lead to monotonic deductions – also given by him – that are very easy to apply

## 10.14. Refutation of false statements on functional dependencies

- Example 3.  $X \rightarrow Y$  cannot be deduced from  $XA \rightarrow YA$ .
  - Proof. The following relation satisfies  $BA \rightarrow CA$  but does not satisfy  $B \rightarrow C$ .

| A | B | C |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 3 | 5 |

- Some observations
  - Note that  $BA \rightarrow CA$  holds “vacuously”; the argument is sound.
  - Advice: Always, try short examples to “dis-” prove something

## 10.15. Closure of a set of functional dependencies

- Suppose  $\mathcal{F}$  is a set of functional dependencies
  - Then  $\mathcal{F}^+$ , called the *closure* of  $\mathcal{F}$ , denotes the set of all functional dependencies that can be deduced from  $\mathcal{F}$ .
  - From completeness of Maier's rules it follows that  $\mathcal{F}^+$  is precisely the set of all functional dependencies that are implied by  $\mathcal{F}$ .
- It is not productive to enumerate a closure
  - As an illustration consider a small example
$$\mathcal{F} = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
  - There are 7 non-empty subsets of ABC
  - Hence, there are  $7 * 7 = 49$  possible dependencies over ABC
  - In this case all of them will show up in  $\mathcal{F}^+$
- Uses of the closure notation
  - We proved “ $A \rightarrow CE$  can be deduced from  $A \rightarrow BC$  and  $B \rightarrow DE$ ”  
This can also be stated as “ $A \rightarrow CE$  is in  $\{A \rightarrow BC, B \rightarrow DE\}^+$ ”
  - We could also say that  $C \rightarrow E$  is not in  $\{A \rightarrow BC, B \rightarrow DE\}^+$
  - If functional dependencies in  $\mathcal{F}$  are constraints associated with a database then  $\mathcal{F}^+$  is the set of *all* functional dependencies that are satisfied in the all legal states of the database
- An interesting example
  - $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}^+ = \{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}^+$
  - This also implies that as constraints,  $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$  or  $\{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}$  will lead to the same legal states of a database over ABC
- With this background, we return to the three desirable properties of a database schema

## 10.16. Losslessness

- $(R_1, R_2, \dots, R_n)$  is a *decomposition* of  $R$  if  $R = R_1 \cup R_2 \cup \dots \cup R_n$ .
- Examples
  - $(AB, BC, DEF)$  is a decomposition of  $ABCDEF$
  - $(AB, BE, DEF)$  is not a decomposition of  $ABCD$  or  $ABCDEF$
- We state a theorem without proof
  - Theorem. Decomposition  $(R_1, R_2)$  is lossless if and only if at least one of the following functional dependencies holds
    - $R_1 \cap R_2 \rightarrow R_1$
    - $R_1 \cap R_2 \rightarrow R_2$
- Example. With  $S \rightarrow A$ ,  $SI \rightarrow P$ ; the decomposition  $(SA, SIP)$  of  $SAIP$  is lossless.
  - Proof. In the decomposition  $(SA, SIP)$ ,  $R_1 = SA$ ,  $R_2 = SIP$ ,  $R_1 \cap R_2 = S$   
 $R_1 \cap R_2 \rightarrow R_1$  is  $S \rightarrow SA$  which holds  
Hence by the above theorem the decomposition  $(SA, SIP)$  is lossless
- Example. With  $S \rightarrow A$ ,  $SI \rightarrow P$ ; decomposition  $(SI, AIP)$  of  $SAIP$  is lossy.
  - Proof.  $R_1 \cap R_2 \rightarrow R_1$  is  $I \rightarrow SI$  which does not hold.  
 $R_1 \cap R_2 \rightarrow R_2$  is  $I \rightarrow AIP$  which does not hold either.  
Hence, the decomposition  $(SI, AIP)$  is lossy.

## 10.17. Dependency preservation

- We are given a database design
  - The design consists of a set of relational schemas and dependencies
  - We want to enforce dependencies through updates on relations in the database
- Example. Does the design (SA, SIP) facilitate preservation of  $S \rightarrow A$ ,  $SI \rightarrow P$ ?

$S \rightarrow A$  can be enforced in SA, without having to look at SIP

$SI \rightarrow P$  can be enforced in SIP, without having to consider SA

Hence, (SA, SIP) facilitates preservation of  $S \rightarrow A$ , and  $SI \rightarrow P$ .

- Example. Does (AB, BC) facilitate preservation of  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow A$ ?

$A \rightarrow B$  can obviously be enforced when AB is updated

Similarly,  $B \rightarrow C$  can be enforced in when BC is updated

It is not immediately obvious if  $C \rightarrow A$  can be preserved, but the answer is that it *can* also be preserved

To see this, recall

$$\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}^+ = \{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}^+$$

$A \rightarrow B$ ,  $B \rightarrow A$ ,  $B \rightarrow C$ ,  $C \rightarrow B$  can be preserved in (AB, BC)

Hence,  $A \rightarrow B$ ,  $B \rightarrow A$ ,  $B \rightarrow C$ ,  $C \rightarrow B$  can be preserved in (AB, BC)

Hence  $\{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}^+$  can be preserved in (AB, BC)

Hence  $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}^+$  can be preserved in (AB, BC)

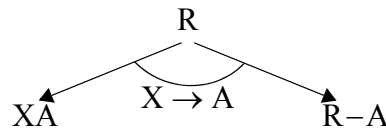
Hence  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow A$  can be preserved in (AB, BC)

## 10.18. Redundancy and functional dependencies

- Suppose  $r$  is a relation. A set of attributes is called a *superkey* of  $r$  if it includes a key of  $r$ 
  - Recall  $SI$  is a key of  $saip$  (SAIP). As  $SI$  is a superset of  $SI$ ,  $SI$  is also a superkey of  $saip$ .  $SIA$ ,  $SIP$  and  $SAIP$  are also superkeys.
- Recall that in SAIP,  $S \rightarrow A$  causes redundancy
- In general we ask when  $X \rightarrow A$  that holds in  $R$  can cause redundancy in  $R$ ?
- It's easier to answer when  $X \rightarrow A$  *cannot* cause redundancy?
  - Here are two cases:
    - (a) When  $X \rightarrow A$  is trivial  
For example,  $A \rightarrow A$  or  $SA \rightarrow A$  cannot cause redundancy
    - (b) When  $X$  is a superkey of  $r$   
For example,  $SI \rightarrow P$  cannot cause redundancy in  $saip$   
Likewise,  $SAI \rightarrow P$  cannot cause any redundancy
- Only causes of redundancy are when neither (a) nor (b) is true
  - For example, return to  $S \rightarrow A$ . It causes redundancy because it is not a trivial dependency and its left hand side does not contain the key  $SI$ .

## 10.19. Boyce Codd Normal Form (BCNF)

- A relation schema in BCNF is meant to ensure absence of redundancy
  - Suppose  $r$  is a relation over  $R$ . Then we say that  $R$  is in *BCNF* if the following if whenever  $X \rightarrow A$  is a functional dependency that holds in  $R$  then either (a)  $X \rightarrow A$  is trivial, or (b)  $X$  is a superkey of  $R$
  - A database schema is said to be in *BCNF* if schema of every relation in the database is in BCNF.
- A violation of BCNF and how to fix it
  - Suppose  $X \rightarrow A$  holds in  $R$ . Then  $X \rightarrow A$  is a violation of BCNF in  $R$  if  $X \rightarrow A$  is non-trivial and  $X$  is not a superkey in  $R$
  - This violation is eliminated by decomposing  $R$  into  $(XA, R-A)$
  - This is shown graphically as follows

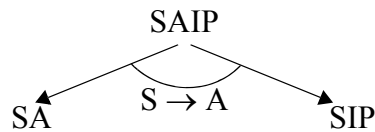


- Caution: there may be other violations of BCNF in  $XA$  and  $R-A$   
Violations have to be eliminated recursively to arrive at a BCNF decomposition



## 10.20. BCNF decomposition in our running example

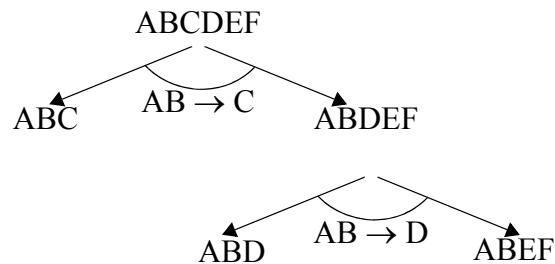
- Consider SAIP with  $S \rightarrow A$ , and  $SI \rightarrow P$ .
  - SAIP is not in BCNF; first, we prove  $S \rightarrow A$  violates BCNF
- Keys and superkeys of SAIP
  - Key(s): SI
  - Superkeys: SI, SIA, SIP, SAIP
- $S \rightarrow A$  is a violation of BCNF in SAIP
  - $S \rightarrow A$  is not trivial and S is not a superkey of SAIP
- Next, we fix this violation by decomposing into (SA, SIP)



- Prove SA is in BCNF
  - Only nontrivial dependency  $S \rightarrow A$  holds in SA
  - Key(s): S is the key in SA; S and SA are the superkeys in SA
  - $S \rightarrow A$  is not a violation of BCNF in SA because S is a superkey in SA
- Prove SIP is also in BCNF
  - Left as an exercise
- Therefore, (SA, SIP) is a BCNF decomposition

## 10.21. BCNF decomposition: Another example

- Consider ABCDEF with  $AB \rightarrow CD$ ,  $CE \rightarrow F$ ,  $F \rightarrow ABE$ 
  - Keys are ABE, CE, and F; superkeys are supersets of these
  - Simplify the FDs:  $AB \rightarrow C$ ,  $AB \rightarrow D$ ,  $CE \rightarrow F$ ,  $F \rightarrow A$ ,  $F \rightarrow B$ ,  $F \rightarrow E$
  - None of  $F \rightarrow A$ ,  $F \rightarrow B$ ,  $F \rightarrow E$ ,  $CE \rightarrow F$  violate BCNF in ABCDEF
  - $AB \rightarrow C$ ,  $AB \rightarrow D$  are violations of BCNF in ABCDEF
  - We choose  $AB \rightarrow C$  among these arbitrarily
  - (Different choices can lead to different BCNF decompositions)
  - To fix the violation, we decompose ABCDEF into (ABC, ABDEF)
  - ABC is in BCNF
  - Now consider ABDEF.
    - Keys are ABE and F; superkeys are supersets of these.
    - $AB \rightarrow D$  is a violation of BCNF
    - Decompose ABDEF into (ABD, ABEF)
    - Both ABD and ABEF are in BCNF
  - We have the decomposition (ABC, ABD, ABEF)
  - Following is intuitive diagrammatically



- We could hand improve this to (ABCD, ABEF)

## 10.22. Problems with BCNF

- BCNF decomposition preserves losslessness (that is good)
- But, BCNF decomposition does not always preserve dependencies
- Example: Consider CAZ relation
  - $C$  = City (together with a state),  $A$  = StreetAddress,  $Z$  = Zipcode
  - $Z \rightarrow C, CA \rightarrow Z$
  - Keys are  $CA$  and  $AZ$
  - $Z \rightarrow C$  is a BCNF violation leading to decomposition ( $CZ, AZ$ )
    - The decomposition is lossless
    - This is the only BCNF decomposition
  - But  $CA \rightarrow Z$  can not be preserved via  $CZ$  or via  $AZ$
- This means that following cannot coexist
  - BCNF level redundancy removal through functional dependencies
  - Losslessness
  - Dependency preservation
- What if we can we live with some redundancy?
  - Third Normal Form (3nf), leaves some redundancy
  - A third normal form lossless decomposition that also preserves dependencies is always possible

### 10.23. Third Normal Form: prime attributes

- An attribute in  $R$  is *prime* if it is in some key of  $R$ 
  - Example: Consider SAIP with  $S \rightarrow A$  and  $SI \rightarrow P$   
The only key is  $SI$ ;  
Here  $S$  and  $I$  are prime, and  $A$  and  $P$  are non-prime
- Suppose  $X \rightarrow A$  is not trivial and holds in  $R$ , then  $X \rightarrow A$  is a violation of ...
  - ... BCNF in  $R$  if  $X$  is not a superkey in  $R$
  - ... 3NF in  $R$  if  $X$  is not a superkey in  $R$  or  $A$  is prime
- A schema with FDs is in 3NF if there is no violation of 3NF in it (the closure of FDs)
- Recall the SAIP example
  - $S \rightarrow A$  is a violation of (BCNF as well as) 3NF
  - $(SA, SIP)$  is a BCNF decomposition  
It is easily seen to be a 3NF decomposition as well  
Shortly we will also derive this 3NF decomposition algorithmically
- Recall CAZ with  $Z \rightarrow C$ ,  $CA \rightarrow Z$ 
  - Keys are  $CA$  and  $AZ$ , hence all attributes ( $CAZ$ ) are prime
  - $Z \rightarrow C$  is a BCNF violation, but it is not a 3NF violation
  - $(CZ, AZ)$  is a BCNF decomposition,
  - $CAZ$  is already in 3NF – no decomposition is needed – and obviously it is lossless and preserves FDs
- Next, we give an algorithm for 3NF decomposition that is lossless and preserves dependencies

## 10.24. Algorithm to obtain 3NF decomposition

- We need to define minimal sets of FDs
- A set of FDs  $F$  over  $R$  is *minimal* if the following conditions are satisfied
  - Every FD in  $F$  is of the form  $X \rightarrow A$ , where  $A$  is a single attribute
  - For no FD  $X \rightarrow A$  in  $F$  is  $F - \{X \rightarrow A\}$  equivalent to  $F$
  - For no FD  $X \rightarrow A$  and proper subset  $Y$  of  $X$  is  $F - \{X \rightarrow A\} \cup \{Y \rightarrow A\}$  equivalent to  $F$
- Algorithm for lossless dependency preserving decomposition in 3NF
  - Input: A schema  $R$  with a set of functional dependencies  $F$   
Without loss of generality we assume  $F$  is a minimal cover  
We also assume that every attribute in  $R$  participates in some FD in  $F$
  - Procedure:
    - if some FD in  $F$  involves all attributes of  $R$ , output  $R$  and return
    - else for every group of FDs  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$  in  $F$  that share the same left hand side, output  $XA_1A_2\dots A_n$
    - For the remaining FDs  $X \rightarrow A$  in  $F$  output  $XA$
    - If some component is a proper subset of another, remove the former
- Examples
  - For  $R = SAIP$  and  $F = \{S \rightarrow A, SI \rightarrow P\}$ , output the decomposition (SA, SIP)
  - For  $R = CAZ$ ,  $F = \{Z \rightarrow C, CA \rightarrow Z\}$  the algorithm will first output (CZ, CAZ), but only (CAZ) will remain as it is a proper superset of CZ
  - For  $R = ABCDEF$ ,  $F = \{AB \rightarrow CD, CE \rightarrow F, F \rightarrow ABE\}$  output the decomposition (ABCD, CEF, ABEF)

## 10.25. Other dependencies

- There are other dependencies, e.g., multivalued dependencies, that explain additional forms of redundancies
  - We have only covered functional dependencies

## Chapter 11

### *The XML model for information*

- XML syntax
  - Elements and attributes
  - Text and tree-based views
  - Why XML is efficient
- Examples for uses of XML
  - How XML can absorb relational model
  - Realizing the world wide web as an XML document
- Subsets of XML documents as collections
  - Concept of axis in XML
  - XPath – a language for drawing subsets from an XML documents
  - The subsets are counterpart of relations in SQL
  - XQuery facilitates queries of thees subsets  
(XQuery is covered in the next chapter)

## 11.1. Semistructured Data Model

- Structured data models
  - Relational and object-oriented approaches deal with structured data
  - Objects (or tuples) in a collection must have a uniform type (schema)  
All objects of a given type must have same properties  
The only leeway is that some properties may have null values
- Lack of structure in the real world
  - Every person may not have a phone
  - House addresses organized differently in different countries
  - Name may be a string or a structure (FirstName, MI, LastName)
  - Two biological species may have (similarities and) differences
- (Totally) structured vs. (totally) unstructured data
  - Models and languages for structured data would be more efficient than those for unstructured data
  - Structured data has a schema associated with it
  - Unstructured data does not have a pre-specified schema
- Semistructured data
  - Middle-ground between structured and unstructured data
  - Can lean on structured side and have a schema
  - Can lean on unstructured side and have no pre-specified schema  
The schema and data must be interleaved together



## 11.2. XML document: a quick example

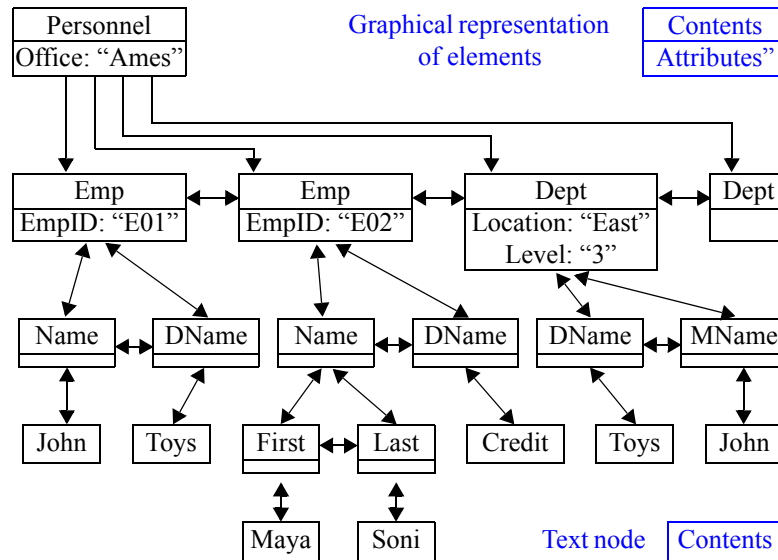
- An example covering the core syntax of XML

```
<PersonnelDB Office="Western">
 <Emp EmpID="E01">
 <Name>John</Name>
 <DName>Toys</DName>
 </Emp>
 <Emp EmpID="E02">
 <Name> <First>Maya</First> <Last>Soni</Last> </Name>
 <DName>Credit</DName>
 </Emp>
 <Dept Location="San Jose" Level = "3" >
 <DName>Toys</DName>
 <MName>John</MName>
 </Dept>
 <Dept Location="Ames">
 </Dept>
</PersonnelDB>
```

- Elements and attributes are the core building blocks
  - One Personnel element containing two Emp and two Dept elements
  - Attributes are Office, EmpID, Location and Level
  - <Dept Location="Ames"> </Dept> is an *empty* element  
Another way of writing it is <Dept Location="Ames"/>
  - We will ignore white spaces, e.g. Level = "3" is same as Level="3"
- The examples covers core syntax of XML ...
  - Although there is more to the syntax - but this forms a sound basis
  - Other features include comments, processing instructions, CDATA etc.
  - We cover name spaces shortly

## 11.3. A tree-based view of a document

- Graphical tree-based representation



- The text-based document of previous page is seen as a tree of nodes
- Prospects for success of XML
  - Storage structured can be broadly divided into three groups: linear, trees, and graphs
  - Trees generalizes sequences and graphs generalize trees
  - For mapping real-world applications, linear structures are most rigid, trees are far more flexible, and graphs are the ultimate
  - Linear and tree structures are efficient but graphs may not be
  - Being tree-based, XML is a natural choice for many applications and efficiency is reassuring

## 11.4. Miscellaneous issues

- Elements vs. attributes

- The following representation is element-centric

```
<Catalog Currency="Singapore dollar">
 <Item> <Name>Nut</Name> <Price>1.00</Price> </Item>
 <Item> <Name>Bolt</Name> <Price>0.50</Price> </Item>
</Catalog>
```

- The following makes greater use of attributes

```
<Catalog Currency="Singapore Dollar">
 <Item Name="Nut" Price="1.00"/>
 <Item Name="Bolt" Price="0.50"/>
</Catalog>
```

- A useful analogy: elements are nouns and attributes are adjectives
- Elements can be nested and subjected to further complex evolution
- Currency is *perhaps* best left as attribute

- Reserved characters

<	>	&	'	"
&lt;	&gt;	&amp	&apos;	&quot;

- A legal XML document: [Prolog] Body [Epilog]

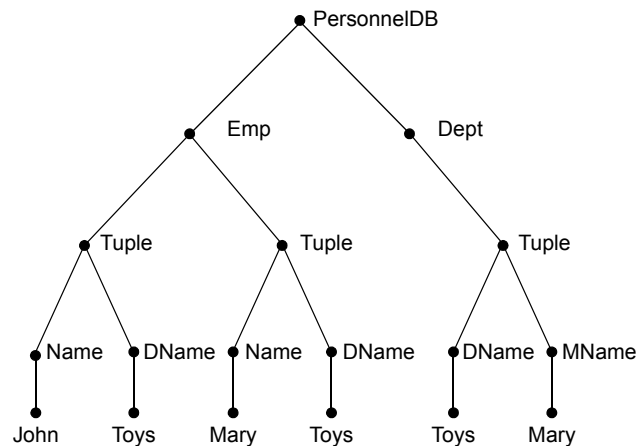
- Body, an element tree, is required; Prolog and Epilog are optional
- Prolog gives information about the document such as XML version, character set used, comments, validating schema, name space abbreviations, processing instructions, etc.
- Epilog contains more comments, processing instructions, etc.

## 11.5. Namespaces in XML documents

- Covered FYI, not included in the course
- Namespaces provide a framework to eliminate ambiguity at global scale
- For example, “DName” in our PersonnelDB may be used by someone else
  - We may use “CS363:DName” to avoid confusion
  - The prefix “CS363” may not guarantee uniqueness globally
  - The URL “http://www.cs.iastate.edu/~cs363:DName” will guarantee uniqueness
  - Syntactically correct URLs are required to specify a namespace
- The name space “http://www.cs.iastate.edu/~cs363”
  - Consists of the following names: PersonnelDB, Emp, Name, DName, Salary, Dept, MName, Office, Location, etc.
  - Objects in our PersonnelDB cannot be confused with any other objects on this globe.
- Declaring namespace and using them
  - Abbreviated names can be declared and used as prefixes  
xmlns: CS363="http://www.cs.iastate.edu/~cs363"
  - Such declaration can be put in a prolog
  - CS363:DName, etc., can be used in the scope of declaration
- Suppose DName denoted document name in some context
  - xmlns: docs="http://www.bits.in"
  - docs:DName and CS363:DName cannot be confused within the scope of declaration of docs and CS363 namespaces

## 11.6. Semistructured data model and XML

- A model for semistructured data
  - Introduced in mid 1996 by Buneman, Davidson, Hillebrand, Suciu
  - The model views data as an edge-labeled tree



- The above has same info as the following Personnel database:

Emp		Dept	
Name	DName	DName	MName
John	Toys	Toys	Mary
Mary	Toys		

- The model is a generalization of relational databases
- A tree recognizes hierarchy in data
- XML can also be used to represent hierarchical data
- The development of semistructured data model + query language have been absorbed into XML + XQuery

## 11.7. XML documents for PersonnelDB

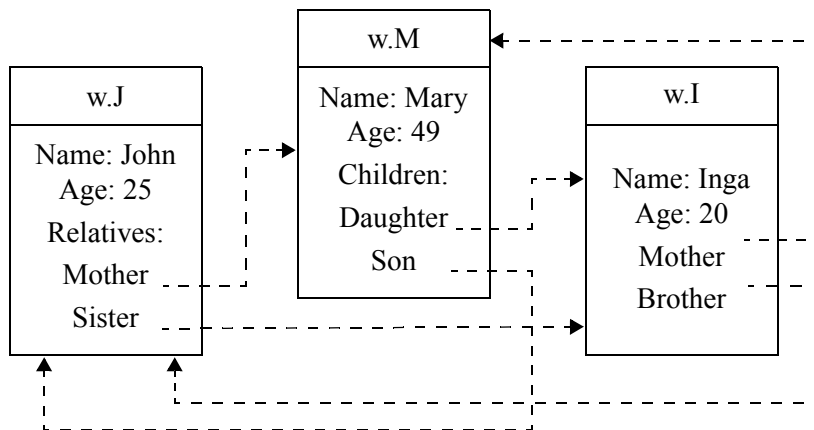
- XML representation of PersonnelDB on previous page

```
<PersonnelDB>
 <Emp>
 <Tuple>
 <Name>John</Name>
 <DName>Toys</DName>
 </Tuple>
 <Tuple>
 <Name>Mary</Name>
 <DName>Toys</DName>
 </Tuple>
 </Emp>
 <Dept>
 <Tuple>
 <DName>Toys</DName>
 <MName>Mary</MName>
 </Tuple>
 </Dept>
</PersonnelDB>
```

- An XML document is hierarchical and always well formed
- It consists of nested elements
  - This document has one PersonnelDB element
  - PersonnelDB element contains Emp and Dept elements
  - Emp element contains two Tuple elements, and so on

## 11.8. Websites may be realized as a database

- The following shows three websites
  - Taken from Abiteboul, Buneman, Suciu book



- A hypothetical XML representation

```
<theWeb>
 <Website Link="w.J">
 <Name>John</Name>
 <Age>25</Age>
 <Relatives>
 <Mother Link = "w.M"/>
 <Sister Link = "w.I"/>
 </Relatives>
 </Website>
 <Website Link="w.M"> ... </Website>
 <Website Link="w.I"> ... </Website>
</theWeb>
```

- Since we can query XML documents ...
  - We could potentially query the world wide web for content

## 11.9. The axis concept in XML

- An axis is a linearized subset of nodes in an XML document
  - Presumably, axis-based subsets are logical to users
  - The linearization requirement assures efficient stream-based processing
  - High level code for traversal of an axis is easy to produce
- W3C recommends required inclusion of 13 axes (abbreviated syntax shown when available)
  - child (/), descendant (//)
  - parent (..), ancestor
  - following-sibling, preceding-sibling
  - following, preceding
  - attribute (@)
  - namespace
  - self (.)
  - descendant-or-self; ancestor-or-self
- Other axes are possible as well
  - Theoretically number of axes is rather huge



## 11.10. XPath expressions

- XPath expressions
  - An XPath step is of the form “axis[predicate]”  
It returns a set of nodes in the axis that satisfy the given predicate
  - An XPath expression is a sequence of XPath steps  
It always returns a set of nodes; the set can be empty, a singleton, or consist of multiple nodes
- For examples of XPath expressions ...
  - Let’s fix the context R as the root node PersonnelDB – see Page 155
- Examples of XPath expressions relative to R
  - /Emp returns two Emp child elements of R
  - /Emp[2] returns a singleton containing the second Emp child element
  - /Emp[last()] returns the last Emp, same as Emp[2] in this case
  - /Emp/@EmpID is a 2-step XPath expression that returns “E01”, “E02”  
Note that in @EmpID an attribute is prefixed with “@”
  - /Emp[Name="John"] returns the Emp element of John
  - /Emp[Name="John"]/. returns the same node as above
  - /Emp[Name="John"]/.. returns the parent, node R
  - /This returns an empty result – not to be interpreted as no result
  - /Emp/Name, a 2-step XPath expression, returns two Name element grand children of R
  - //DName returns the three DName descendants of R  
In general “//” will select all descendants irrespective of their level  
This makes “//” versatile – somewhat independent of structure of R
  - /\* returns all four children of R – two Emp and two Dept elements
  - //MName/text() returns the string “Mary”
  - /Dept[@Location and @Level] returns Dept having both attributes

### 11.11. XPath expression offer high-level building block

- (Taken from an IBM website:) Consider the XPath expression

`//book[author="Neal Stephenson"]/title`

- It seem to match “books authored by Neal Stephenson”, perhaps in the context of a library catalog
- Without the concept of XPath expressions the following code may be needed (written in Java with DOM API)

```
ArrayList result = new ArrayList();
NodeList books = doc.getElementsByTagName("book");
for (int i = 0; i < books.getLength(); i++) {
 Element book = (Element) books.item(i);
 NodeList authors = book.getElementsByTagName("author");
 boolean stephenson = false;
 for (int j = 0; j < authors.getLength(); j++) {
 Element author = (Element) authors.item(j);
 NodeList children = author.getChildNodes();
 StringBuffer sb = new StringBuffer();
 for (int k = 0; k < children.getLength(); k++) {
 Node child = children.item(k);
 // really should to do this recursively
 if (child.getNodeType() == Node.TEXT_NODE) {
 sb.append(child.getNodeValue());
 }
 }
 if (sb.toString().equals("Neal Stephenson")) {
 stephenson = true;
 break;
 }
 }
 if (stephenson) {
 NodeList titles = book.getElementsByTagName("title");
 for (int j = 0; j < titles.getLength(); j++) {
 result.add(titles.item(j));
 }
 }
}
```

- The complexity signature of Java code can be expressed as `{{{{{{}}}}}}`, where each `{}` represents a loop or a conditional
- Obviously, the XPath expression is far simpler and it may work even for a consortium of libraries
- Hypothesize an assignment statement “almost” as simple as  
“`x := Catalog//book[author="Neal Stephenson"]/title;`” ...  
...to imagine the power of XML – and this is just a start

## 11.12. The role of XPath in XML technologies

- In W3 Consortium's view XPath is a basic primitive
  - (Most) processing technologies for XML take this for granted  
This includes DOM, XSLT, and XQuery
- An XML document is a rich repository of collections
  - Collections are semantic counterpart of path expressions
  - XQuery helps in drawing further subsets of these collections
  - XQuery will be covered in the next chapter
  - Similar observations can be made about other processing technologies

## 11.13. Schema based validation

- In information one thinks in terms of instance and schema
  - The contents are governed by an instance
  - And the structure is governed by a schema
  - Whereas instance change with time, schema remains stable
  - For example, a relational database has a state and a schema
- In XML too one thinks in terms of ...
  - ... an instance document and a schema
  - There are two major types of schemas: DTD (Document Type Definition) and XML Schema
  - XML Schemas are a far more versatile option than DTD
- XML Schema
  - An XML Schema is itself an XML document, typically with a file with .xsd extension in its name
  - XML Schema provides a vast array of built-in elementary types
  - Users can create complex types
  - sequence and choice are two important built-in type constructors
  - A schema even validates how an instance document is organized
- Associating an instance with a schema
  - The location of schema is included in the header element of instance as a value of attribute SchemaLocation  
For example, SchemaLocation = "www.iastate.PersonnelDB.xsd"
- Example of schema
  - We defer a detailed example to Chapter 13

## Chapter 12

### *XQuery: the query language for XML documents*

- XQuery is a query language for XML
  - It is a W3 recommended standard  
Official W3 link – [www.w3.org/TR/xquery/](http://www.w3.org/TR/xquery/)
- A language for expressions
  - Arithmetical, boolean, XPath, FLWOR expressions, etc.
- We mainly concentrate upon the FLWOR expressions
  - FLWOR stand for clauses for, let, where, return, and order by  
... counterpart of select, from, where, group by, and having of SQL
- References
  - A general tutorial – [www.w3schools.com/xquery/default.asp](http://www.w3schools.com/xquery/default.asp)
  - For functions in XQuery –  
[www.w3.org/2005/xpath-functions/](http://www.w3.org/2005/xpath-functions/)  
[www.w3schools.com/xquery/xquery\\_functions.asp](http://www.w3schools.com/xquery/xquery_functions.asp)  
[www.w3schools.com/xpath/xpath\\_functions.asp](http://www.w3schools.com/xpath/xpath_functions.asp)
- Acknowledgment
  - Some examples are taken from W3 web pages for XQuery

## 12.1. XQuery

- XQuery is a language for expressions
  - Arithmetic, boolean, XPath, FLWOR, etc.
  - $10 * 5$   
returns 50.0
  - if ( $10 * 5 = 50$ ) then 55 else 56  
returns 55.0
  - We will mainly concentrate on FLWOR expressions
  - FLWOR expressions include XPath expressions as subexpressions  
XPath is a language in its own right
  - XQuery is a strongly typed language; however it allows type casting
- Input and output in SQL vs. XQuery
  - In SQL queries, input as well as output are relations
  - Inputs to an XQuery query are XML documents
  - But the output is not required to be an XML document  
... it can be an XML document, plain text, an html document, etc.
  - This is to help yield ready-to-use information to applications
- Notes about style of presentation
  - We take freedom in adding / removing white spaces in output

## 12.2. FLWOR expressions in XQuery

- A FLWOR expression in XQuery has the following form
  - One or more for and let clauses
    - [where clause]
    - [order by clause]
    - return clause
- for and let clauses
  - These clauses introduce variables for path expressions
  - Path expressions bind to collections of nodes (see Chapter 11)
  - This sets up an environment for where-order by-return clauses
    - ... for clause creates an iteration serving one variable binding at a time
    - ... let clause binds a variable at once to all matches of path expression
- where clause
  - The where clause applies to bindings from all for and let clauses
  - It can be considered a counterpart of where + having clauses in SQL
  - In SQL where clause is applied before having clause
    - This means information (tuples) removed by where clauses are not available to having and select clauses
    - But the where clause of XQuery sees all information
- order by clause is similar to that in SQL
  - Ordering can only be done for atomic values – not arbitrary paths
- return clause can only return on node
  - If only one node N is to be returned – it is simple
  - For one or more nodes to be returned “return <R> {N, N, ..., N}</R>” can to be used, where R is a user-defined tag, and {.} forces evaluation
  - Without {.} <R> {N, N, ..., N}</R> is returned as a literal string

### 12.3. A basic example of FLWOR expressions (2 slides)

- FLWOR expressions are counterpart of SQL's select
  - FLWOR is pronounced “flower”  
It stands for the for, let, where, order by and return clauses
  - We consider a motivating example in SQL and XQuery
- Example of an input, a query, and an output in SQL
  - Input – Emp(Name, DName, Salary) relation
  - An SQL query – select e.Name from Emp e where e.DName = 'Toys'
  - Output – the following relation:

Name
John
Mary

- Counterpart in XQuery
  - Input – Emp.xml XML document:

```
<Emps>
 <emp> <Name>John</Name>
 <DName>Toys</DName>
 <Salary>50000</Salary>
 </emp> ...
</Emps>
```

Caution: Emp.xml is a document, Emps is the root element, emp is an element
  - XQuery query:

```
for $e in doc("Emp.xml")/Emps/emp
where $e/DName/text() = 'Toys'
return $e/Name
```
  - The output:

```
<Name>John</Name>
<Name>Mary</Name> ...
```



## 12.4. A basic example of FLWOR (Slide 2 of 2)

- Returning multiple values via return clause
  - Consider returning names as well as salaries of employees

```
for $e in doc("Emp.xml")/Emps/emp
where $e/DName/text() = 'Toys'
return $e/Name, $e/Salary
```
  - The return clause with more than one element is syntactically incorrect – surround with a (any) tag to fix it

```
for $e in doc("Emp.xml")/Emps/emp
where $e/DName/text() = 'Toys'
return <NS> {$e/Name, $e/Salary} </NS>
```
  - Here, "{...}" is needed to force evaluation  
Also, only one expression or one element can be returned
  - Query returns the following

```
<NS> <Name>John</Name> <Salary>50000</Salary> </NS>
<NS> <Name>Mary</Name> <Salary>60000</Salary> </NS>
```
- Add more features ...
  - The result is not an xml document; if desired, add a (any) root element
  - Return the result sorted by name in descending order

```
<ToysSalaries> {
 for $e in doc("Emp.xml")/Emps/emp
 where $e/DName/text() = 'Toys'
 order by ($e/Name) descending
 return <NS> {$e/Name, $e/Salary} </NS>
} </ToysSalaries>
```
  - Query returns the following

```
<ToysSalaries>
 <NS> <Name>Mary</Name> <Salary>60000 </Salary> </NS>
 <NS> <Name>John</Name> <Salary>50000</Salary> </NS>
</ToysSalaries>
```

## 12.5. How for and let clauses work

- Consider the following document (RABC.xml) and query

```
<R>
 <A>
 <C>1</C> <C>1</C> <C>1</C> <C>1</C>
 <C>2</C> <C>2</C> <C>2</C> <C>2</C>
 <A>
 <C>3</C> <C>3</C> <C>3</C> <C>3</C>
 <C>4</C> <C>4</C> <C>4</C> <C>4</C>
 <A>
 <C>5</C> <C>5</C> <C>5</C> <C>5</C>
 <C>6</C> <C>6</C> <C>6</C> <C>6</C>
</R>
```

```
for $a in doc("RABC.xml")/R/A
for $b in $a/B
let $c := $b/C
where
return sum($c)
```

- The query returns "4 8 12 16 20 24"

- Explanation

- Any number of for and let clauses in any sequence can precede a given pair of where and return clauses
- A for-variable binds to one node at a time creating an iteration  
A let-variable binds at-once to all nodes it matches
- Here, \$a and \$b in for clause create  $2 * 3 = 6$  iterations  
Each iteration creates environment consisting of 1 \$a, 1 \$b, 4 \$c nodes
- In this case the where clause is empty, and does not eliminate any of the 6 iterations
- Each iteration returns sum of 4 \$c values (<C>n</C> is coerced to n)
- Remark: A single "for \$a in doc("RABC.xml")/R/A, \$b in \$a/B" could be used instead of the two for clauses

## 12.6. Aggregates in FLWOR expressions

- Consider an SQL example

```
select e.DName, sum(e.Salary)
from Emp e
where e.Salary >= 100000
group by e.DName
having count(*) >= 10
```

- Where clause removes all employees earning below 100,000  
Removed ones are invisible to group by, having, and select clauses
  - We cannot sum the salaries of *all* employees in a department
  - The problem cannot be fixed by a minor modification in SQL query;  
one has to think of a new way of writing it
  - This situation has been improved in XQuery
- Consider the following XQuery expression ...

```
for $d in distinct-values(doc("Emp.xml")/Emps/emp/DName)
let $s1 := doc("Emp.xml")/Emps/emp[DName = $d]/Salary
let $s2 := doc("Emp.xml")/Emps/emp[DName = $d
 and Salary >= 100000]/Salary
where count($s2) >= 10
return <DS> {$d, <SumSal> {sum($s1)} </SumSal>} </DS>
```

- Explanation

- “distinct-values()” prevents a value from appearing multiple times
- For a department two parallel groups \$s1 and \$s2, are created  
\$s1 holds Salary-values of all employees in the department  
\$s2 holds only Salary-values that exceed 100,000  
Whereas \$s2 is used to qualify a department, \$s1 is used in retrieval
- In SQL where only deal with tuples and having only with groups  
having is subservient to where; it can only see tuples allowed by where
- The where clause in XQuery can simultaneously handle elements from  
for clauses as well as sets (“groups”) of elements from let clauses

## 12.7. html output from a FLWOR expression

- An XQuery is not required to return a legal XML document
- The following query returns html code

```
<html>
 <table border = "1">
 <tr>
 <td>Name</td>
 <td>DName</td>
 <td>Salary</td>
 </tr>
 for $e in doc("Emp.xml")//emp
 return
 <tr> {
 <td>{$e/Name/text()}</td>,
 <td>{$e/DName/text()}</td>,
 <td>{$e/Salary/text()}</td>
 } </tr>
 </table>
</html>
```

- In an html browser, the output will display as a table

Name	DName	Salary
Hari	Credit	55000
John	Toys	50000
Leu	Shoes	70000
Mary	Toys	60000
Superman	Audit	100000

- The html output in this case is well formed and technically it can also be considered a legal XML document.
- In general an html document may not be well formed and in such a case it cannot be considered a legal XML document
- This illustrates how XQuery can produce customized results for direct consumption by applications without pre-processing

## 12.8. Restructuring using FLWOR expressions

- Reorganize Emp.xml document by departments

```
<DeptsAndEmps> {
 for $d in distinct-values(doc("Emp.xml")//DName)
 return
 <DeptList> {
 $d,
 <EmpsList> {
 for $e in doc("Emp.xml")//emp[DName=$d]
 return <NameSal> {$e/Name, $e/Salary} </NameSal>
 } </EmpsList>
 } </DeptList>
} </DeptsAndEmps>
```

- The result of this query is shown below.

```
<DeptsAndEmps>
 ...
 <DeptList>
 <DName> Toys </DName>
 <EmpsList>
 <NameSal>
 <Name> John </Name>
 <Salary> 80000 </Salary>
 </NameSal>
 <NameSal>
 <Name> Mary </Name>
 <Salary> 90000 </Salary>
 </NameSal>
 </EmpsList>
 ...
</DeptsAndEmps>
```

- Symmetry between XML and XQuery
  - XML allows nesting at data level and XQuery allows it at query level
  - This achieves symmetry between data and programs

## 12.9. FLWOR expressions: every and satisfies

- XQuery allows the use of every and satisfies quantifiers
- The query *retrieve average salaries in departments where all salaries are 70K or higher* is expressed as follows

```
<SpecialDepartments> {
 for $d in distinct-values(doc("Emp.xml")//DName)
 where every $s in doc("Emp.xml")//emp[DName = $d]
 satisfies ($s/Salary >= 70000)
 return
 <DeptAvg>
 {$d,
 <AvgSal> { avg(doc("Emp.xml")//emp[DName = $d]/Salary)}
 } </AvgSal>
 } </DeptAvg>
} </SpecialDepartments>
```

## 12.10. FLWOR Expressions: querying element tags

- Suppose <Salary> information needs to be suppressed

- Use the name(.) function

```
<Everyone> {
 for $e in doc("Emp.xml")//emp
 return
 <e> {
 for $f in $e/*
 return if (name($f) = "Salary") then "" else $f
 } </e>
} </Everyone>
```

## 12.11. Joins using variations of for and let (2 slides)

- Reporting names of employees and their managers
  - There are many variations and issues
- Here is an XQuery expression

```
<EmpAndManagers> {
 for $e in doc("Emp.xml")//emp
 for $d in doc("Dept.xml")//emp
 where $e/DName = $d/DName
 order by ($e/Name)
 return <EmpManager> {$e/Name, $d/MName} </EmpManager>
} </EmpAndManagers>
```

- The query returns the following xml document

```
<EmpAndManagers>
 <EmpManager>
 <Name> Mary </Name>
 <MName> Mary </MName> </EmpManager>
 <EmpManager>
 <Name> Leu </Name>
 <MName> Leu </MName> </EmpManager>
 <EmpManager>
 <Name> John </Name>
 <MName> Mary </MName> </EmpManager>
</EmpAndManagers>
```



## 12.12. Joins using variations of for and let (Slide 2 of 2)

- Replace inner for by let

```
<EmpAndManagers> {
 for $e in doc("Emp.xml")//emp
 let $m := doc("Dept.xml")//emp[DName = $e/DName]/MName
 order by ($e/Name)
 return <E&M> {$e/Name, $m} </E&M>
} </EmpAndManagers>
```

- The case of employees in departments without managers
  - The inner for fails to match whereas let matches to empty group
  - Previous query (with inner for) does not report such employees
  - This query reports all employees, with or without managers

- The result

```
<EmpAndManagers>
 <E&M> <Name>John</Name> <MName>Mary</MName> </E&M>
 <E&M> <Name>Hari</Name> <MName> </MName> </E&M>
 ...
</EmpAndManagers>
```

- Hari is reported, but without a manager
- In SQL a join condition such as `e.DName = m.DName` would be used  
No information about Hari will survive
- Use let, but exclude employees without managers

```
<EmpAndManagers> {
 for $e in doc("Emp.xml")//emp
 let $m := doc("Dept.xml")//emp[DName = $e/DName]/MName
 order by ($e/Name)
 return if (exists($m)) then <EmpManager> {$e/Name, $m} </EmpManager>
 else ""
} </EmpAndManagers>
```

## 12.13. FLWOR Expressions: ordering

- Ordering is done by default.

```
for $i in (1, 2),
 $j in (3, 4)
return
 <tuple> {
 <i>{ $i }</i>
 <j>{ $j }</j>
 } </tuple>
```

- The output is as follows (formatted for compactness).

```
<tuple> <i>1</i> <j>3</j> </tuple>
<tuple> <i>1</i> <j>4</j> </tuple>
<tuple> <i>2</i> <j>3</j> </tuple>
<tuple> <i>2</i> <j>4</j> </tuple>
```

- Order can be avoided

```
for $i in unordered(1, 2),
 $j in unordered(3, 4)
return
 <tuple> {
 <i>{ $i }</i>
 <j>{ $j }</j>
 } </tuple>
```

- In the output, the tuples may occur in any order.
- Use of “unordered” can make queries more efficient

## 12.14. More on restructuring and ordering

- Consider the following document containing books

```
<bib>
 <book> <title>TCP/IP Illustrated</title>
 <author>W. Stevens</author>
 <publisher>Addison-Wesley</publisher> </book>
 <book> <title>Advanced Unix</title>
 <author>W. Stevens</author>
 <publisher>Addison-Wesley</publisher> </book>
 ...
</bib>
```

- Ordering may be used at multiple levels in a query
- Books with price greater than 100 ordered by 1st author and for each author by title

```
//book[price > 100] order by (author[1], title)
```

- The following returns an alphabetic list of publishers
- Each publisher element contains books, each containing a title and a price, in descending order by price

```
<PublisherList>
 for $p in distinct-values(doc("bib.xml")//publisher)
 order by ($p/name)
 return
 <publisher> {
 <name> {$p/text()} </name>,
 { for $b in doc("bib.xml")//book[publisher = $p]
 order by($b/price) descending
 return <book> {$b/title, $b/price} </book>
 } </publisher>
 }
</PublisherList>
```

## 12.15. FLWOR Expressions: Functions (2 slides)

- The summary function below accepts a sequence of employee elements, summarizes them by department, and returns a sequence of Dept elements.

```
define function summary(element employee* $emps) returns element Dept*
{
 for $d in distinct-values($emps/deptno)
 let $e := $emps[deptno = $d]
 return
 <Dept>
 {$d}
 <headcount> {count($e)} </headcount>
 <payroll> {sum($e/salary)} </payroll>
 </Dept>
}
```

- Using the summary function the following call prepares a summary of employees located in Denver.

```
summary(doc("acme_corp.xml")//employee[location = "Denver"])
```

## FLWOR Expressions: Functions (Slide 2 of 2)

- Recursive and mutually recursive functions are allowed.
  - The following function returns maximum depth of an XML document.

```
define function depth(element $e)
returns xs:integer
{
 {-- An empty element has depth 1 --}
 {-- Otherwise, add 1 to max depth of children --}
 if (empty($e/*)) then 1
 else max(for $c in $e/* return depth($c)) + 1
}
```

- The following is a call to compute depth of partslist.xml.

```
depth(doc("partslist.xml"))
```

- Suppose the filter function below filters A and B elements

- It preserves the order of these elements in the document

```
filter($doc//(A | B)) .
```

This returns a forest.

- The following returns the table of contents from cookbook.xml.

```
<toc>
{filter(doc("cookbook.xml")//
 (section |
 section/title |
 section/title/text()))
}
</toc>
```

## 12.16. FLWOR Expressions: queries on sequence

- XQuery uses the precedes, follows, <<, and >> operators to express conditions based on sequence.
- Consider a surgery report containing procedure, incision, and anesthesia elements.
- The following returns a critical sequence that contains all elements and nodes found between the first and second incisions of the first procedure.

```
<critical-sequence> {
 let $proc := //procedure[1]
 for $n in $proc//node()
 where $n follows ($proc//incision)[1] and $n precedes ($proc//incision)[2]
 return $n
} </critical-sequence>
```

## 12.17. Features of XQuery (2 slides)

- FLWOR expressions are well designed
  - Consider this: in syntax for Java, wherever an integer can be used, an expression that evaluates to an integer can be used too
  - This substitutability makes a language more natural
  - Unlike SQL, XQuery provides a total support for substitutability
  - Over the decades SQL has gone through improvements; but perfect SQL is perhaps no longer possible due to legacy issues
  - Languages must be designed with extreme care; their DNA is inherited by software – that in turn lasts longer than hardware
- Uniform linguistic versatility of elements
  - Whereas a relational database has relations, tuples, and attribute values – having different linguistic behavior, XML only has elements
  - XML and XQuery allow nesting at data and language levels; this helps capture symmetry between data and programs (queries)
- XML / XQuery are high level as well as very low level
  - XML is like an assembly language for data  
At the same time it is a very high level language
  - XQuery too is a very high and low level language
  - We will exhibit this by building object-orientation in XML / XQuery
- Many advanced features of XQuery are not covered here
  - For example run-time type dispatching
- The Query Prolog is a series of declarations and definitions that affect query processing
  - It can be used to define namespaces, import definitions from schemas, and define functions

## 12.18. SQL vs. XQuery

- Database
  - SQL operates on a relational database; the structure is set-centric
  - XQuery operates on XML documents that are sequences of elements with unordered(.) a sequence can be viewed as a set to allow greater optimization
- Schema
  - In case of SQL the database schema is known and fixed
  - XQuery does not require input XML documents to have schema
  - But schemas can be used to validate the document, queries, and optimization
  - Some features such as ID and IDREF require schema awareness
- Data elements
  - In SQL we have to be aware of relations, tuples, and attribute values
  - In XML, everything is an element leading to a symmetry between data and language
- Grouping mechanism
  - In SQL there is a single grouping mechanism, based only on attribute values, and this has to be applied after where clause potentially leading to a loss of context
  - XQuery supports more flexible groups; there can be any number of groups of elements that are parallel to elements with no loss of context
- Query results
  - An SQL query always results in a relation
  - The result of an XQuery is not required to be an XML document, providing greater flexibility in application development



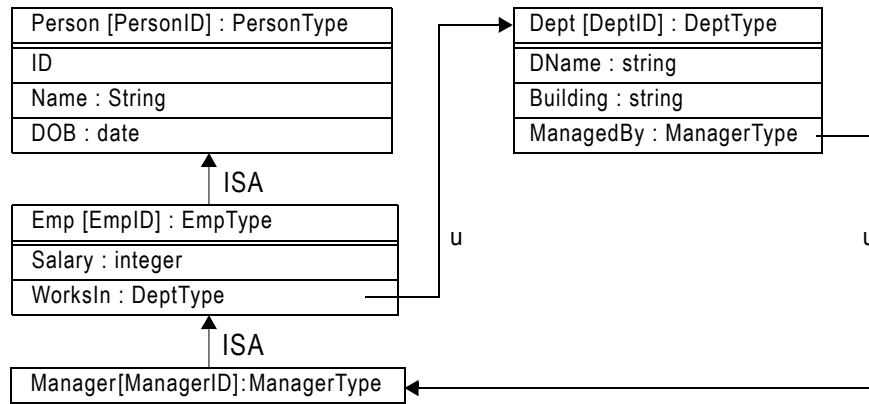
## Chapter 13

### *Object-orientation in XML and XQuery*

- In this chapter we show how to add object-orientation ...
  - ... to XML – giving rise to the dialect OOXML
  - OOXML includes inheritance, references, and paves ground for dotted expressions
  - All XML processing technologies, such as DOM, XPath, and XSLT can benefit from OOXML
  - But our main focus will be on XQuery
- A running example will consist of
  - The instance PersonnelOODB.xml
  - The schema PersonnelOODB.xsd
  - For illustrations we will draw extensively from these documents

## 13.1. Sneak-peek: User view of OOXQuery

- Recall the object oriented schema from Chapter 9



- We have added some details such as “EmpID” and “EmpType”
- Recall the query *what is date of birth of John’s manager*
  - As seen before in Chapter 9, in OQL it is expressed as follows

```

select e.WorksIn.ManagedBy.DOB
from Emp e
where e.Name = 'John'

```
  - In OOXQuery it would be expressed as follows

```

for $e in doc("Personnel_OODB.xml") // Emp
where $e.Name = 'John'
return <MDOB> {$e/WorksIn.ManagedBy.DOB} </MDOB>;

```
  - >>> Ideally, this is all what an OOXQuery user needs to know <<<**
  - A system detail: Internally it is translated into following XQuery query

```

for $e in doc("Personnel_OODB.xml") // Emp
where $e.Name = 'John'
return <MDOB> {
 $e/id($e/id($e/id($e/id($e/WorksIn/@DeptID)
 /ManagedBy/@ManagerID)/As_Emp/@EmpID)
 /As_Person/@PersonID)/DOB/text() } </MDOB>;

```
  - This query can be executed by an XQuery engine

## 13.2. Extension and restriction in XML

- XML offers some high level support for object-orientation ...
  - ... consisting of extension and restriction
- A type can be extended to include additional features
  - For example, to extend PersonType, the base type, to EmpType
  - Salary element is added
- A type can be restricted to exclude some features
  - For example, to restrict EmpType, the base type, to CEOType
  - DName element can be removed
- Extensions and restrictions are useful mechanisms
  - But they do not lead to dotted expressions
  - Users cannot avail syntactic advantages offered by dotted expressions

### 13.3. Low level support for object-orientation in XML

- XML offers the following low-level primitives for object-orientation
  - ID, IDREF, IDREFS type attributes and id(.) function
- ID type attributes for supporting object-identity
  - ID type attribute and value pairs can be included in an element header
  - Example: `<Emp EmpID = "Emp005"> ... </Emp>`  
EmpID is meant to be of ID type and "Emp005" is a string without blanks
  - Validation ensures that all values of all ID type attributes are distinct
  - Thus "Emp005" is unique, and serves as object-identity of the above Emp element, turning it into an "object"
- id(.) function for hopping to an object
  - Following the above example ...  
id("Emp005") returns the Emp object `<Emp EmpID="Emp005">...</Emp>`
- IDREF type attributes for storing pointers to objects
  - Validation ensures that all values of IDREF type that occur in a document indeed point to objects that exist in the document
  - Thus using values of IDREF type as argument of id(.) is "safe"
- IDREFS type attributes for storing pointers to multiple objects
  - A value is a sequence of strings (without blanks) separated by blanks
  - An IDREFS value is treated as a sequence of IDREF values
  - id(.) function returns a sequence of objects for an IDREFS value
- A sequence is a built in type constructor in XML
  - For example, a path expression returns a sequence of elements
  - FYI, another constructor in XML is choice

### 13.4. Problems with low level primitives and our plan

- We use low-level primitives for building OOXML, an object-oriented dialect for XML
- Lack of strong typing in IDREF values
  - Validation ensures that an IDREF value points to an object, but it does not ensure that the object is of a desired type
  - This means that although XML supports types, the type of IDREF or IDREFS values cannot be determined by at compile time
  - In other words, IDREF and IDREFS do not support type safety
- We want object orientation that supports dotted expression
  - The lack of type safety is a serious obstacle
- Our plan ...
  - We will use the low-level primitives to build OOXML
  - Strong typing of objects will be emphasized
  - We will superimpose strong typing for IDREF and IDREFS pointers
  - We will “standardize” representation of inheritance and references
  - The objective is to pave groundwork for including dotted expressions in processing technologies such as DOM, XPath, XQuery, and XSLT
- It is a minimal plan ...
  - Users can mimic inheritance and references by using `id(.)` manually without fear of violating type safety
  - A better alternative is to use dotted expressions that could be translated in terms `id(.)` by a compiler for a processing technology
  - The translation would be based upon a *transition table* that depends upon document schema and independent of a processing technology
  - The expression resulting from translation would be native to a processing technology and executable without any new runtime infrastructure

### 13.5. A quick and partial solution for XQuery

- We have hypothesized OOXQuery, a dialect of XQuery
  - OOXQuery offers object-orientation with inheritance and references
  - Type safety of `id(.)` is guaranteed for objects typed via mechanisms in OOXML (type safety is not guaranteed for other objects)
  - OOXQuery can be expressed with `id(.)` function
  - As stated above, a more attractive alternative for users is to use dotted expressions and rely on the XQuery compiler to translate them in terms of `id(.)`
- The translation of dotted expressions in XQuery queries
  - The ideal would be to incorporate it in XQuery compiler
  - To obtain quick results we have implemented a preprocessor for such translation
- Execution of translated XQuery queries
  - The expression resulting from translation is a native XQuery query
  - Therefore, it can be executed by an XQuery engine

## 13.6. Document organization in OOXML

- In XML organization of an instance is regulated by a schema
- Example
  - The code in PersonnelOODB.xsd that governs the layout in the instance PersonnelOODB.xml instance is as follows

```
<xs:schema ...>
 <xs:element name="PersonnelOODB">
 <xs:annotation>
 <xs:documentation>Personnel OO Schema</xs:documentation>
 </xs:annotation>
 <xs:complexType>
 <xs:sequence>
 <xs:element name="Person" type="PersonType"
 minOccurs="0" maxOccurs="unbounded"/>
 <xs:element name="Emp" type="EmpType" ... />
 <xs:element name="Dept" type="DeptType" ... />
 <xs:element name="Manager" type="ManagerType" ... />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 ...
</xs:schema>
```

- It says that an instance consists of a single PersonnelOODB element
- This element is a sequence of ...
  - a sequence of 0 or more Person elements of type PersonType,
  - followed by a sequence of Emp elements of type EmpType,
  - followed by a sequence of Dept elements of type DeptType,
  - followed by a sequence of Manager elements of type ManagerType
- The dialect OOXML does not propose any changes to how the instance documents are organized
- Alternative organizations are possible
- Thus we mainly need to concentrate on the four types
  - We will consider EmpType in detail

## 13.7. OOXML code level details

- We show some code for our running example
- Code for EmpType in the schema (PersonnelOODB.xsd)

```
xs:complexType name="EmpType">
 <xs:sequence>
 <xs:element name="ISA">
 <xs:complexType>
 <xs:attribute name="oType" type="xs:string" use="required" fixed="PersonType"/>
 <xs:attribute name="oID" type="xs:IDREF" use="required"/>
 </xs:complexType>
 </xs:element>
 <xs:element name="Salary" type="xs:int"/>
 <xs:element name="WorksIn">
 <xs:complexType>
 <xs:attribute name="oType" type="xs:string" use="required" fixed="DeptType"/>
 <xs:attribute name="oID" type="xs:IDREFS" use="required"/>
 </xs:complexType>
 </xs:element>
 </xs:sequence>
 <xs:attribute name="EmpID" type="xs:ID" use="required"/>
</xs:complexType>
```

- EmpType consists one attribute and a sequence of three elements
- The attribute is EmpID, it is of ID type, and its use is required
- The three elements are ISA to capture inheritance, Salary – an int, and WorksIn a reference to an object of type DeptType
- An Emp element object in the instance (PersonnelOODB.xml)

```
<Emp EmpID = "Emp005">
 <ISA oType = "PersonType" oID = "Person006" />
 <Salary>40000</Salary>
 <WorksIn oType = "DeptType" oID = "Dept002" />
</Emp>
```

- It says Emp005 inherits all properties of Person006, has a salary of 40000, and WorksIn department Dept002



## 13.8. Typing of objects and object pointers

- Typing of objects
  - This does not require any new constructs to be added to XML but requires that some conventions are strictly followed
  - Types for objects should be created and enforced
  - For example, Page 193 we have already seen code defining EmpType
  - The code governing the document organization on Page 194 includes  
`<xs:element name="Emp" type="EmpType" ... />`  
It says that all Emp elements are of EmpType
  - The following is included in the root of the code for EmpType  
`<xs:attribute name="EmpID" type="xs:ID" use="required"/>`  
This ensures that all Emp elements will use EmpID for object identity
  - This turns all Emp elements into uniformly defined “objects”
  - Amounts to strongly typing Emp elements in an instance
- Typing of IDREF values
  - Keywords oID and oType are added to OOXML, both are attributes
  - oID is of IDREF (or IDREFS) type; validation ensures that they point to specific object (or sequence of objects)
  - oType is a string whose value is name of a desired type
  - For example: oType = "DeptType" oID = "Dept002" ensures that an object with identity Dept002 exists and that it is type DeptType
- Benefits of strong typing?
  - For example, we may like to say Emp005 works in Dept002
  - If e is an employee, expression e.WorksIn should be of type DeptType
  - Otherwise, we do not know how to process the result and unable to write code that can be validated by a compiler

## 13.9. Adding references

- The EmpType on Page 193 includes following code for WorksIn

```
<xs:element name="WorksIn">
 <xs:complexType>
 <xs:attribute name="oType" type="xs:string" use="required" fixed="DeptType"/>
 <xs:attribute name="oID" type="xs:IDREF" use="required"/>
 </xs:complexType>
</xs:element>
```

- It sets up the object-valued property WorksIn for objects of EmpType
- If variable e is an object of type EmpType, then e.ManagedBy is an object of type DeptType whose oID and oType are required to reside in the WorksIn element of e
- Consider the Emp object on Page 193

```
<Emp EmpID = "Emp005">
 <ISA oType = "PersonType" oID = "Person006" />
 <Salary>40000</Salary>
 <WorksIn oType = "DeptType" oID = "Dept002" />
</Emp>
```

- For this as e, e.WorksIn is the Dept object Dept002 of type DeptType
  - A compiler should translate e.WorksIn as id(e/WorksIn/@oID)
  - The latter can be processed within existing XML
- e/WorksIn/@oID returns the reference to object Dept002 that is guaranteed to be of type DeptType
- id("Dept002") returns the department Dept002 object
- Dept002 can be processed further based upon DeptType
- Obviously type safety is guaranteed

## 13.10. Adding inheritance

- The EmpType on Page 193 includes following code for ISA

```
<xs:element name="ISA">
 <xs:complexType>
 <xs:attribute name="oType" type="xs:string" use="required" fixed="PersonType"/>
 <xs:attribute name="oID" type="xs:IDREF" use="required"/>
 </xs:complexType>
</xs:element>
```

- It says that an object e of type EmpType inherits properties of an object of type PersonType whose oID and oType are required to reside in e
- Consider the Emp object on Page 193

```
<Emp EmpID = "Emp005">
 <ISA oType = "PersonType" oID = "Person006" />
 <Salary>40000</Salary>
 <WorksIn oType = "DeptType" oID = "Dept002" />
</Emp>
```

- For this as e, e.Name has to be found in object Person006 that is of type PersonType
- A compiler should translate e.Name as id(e/ISA/@oID)/Name
- The latter can be processed within existing XML
  - e/ISA/@oID returns the reference to object Person006 that is guaranteed to be of type PersonType
  - id("Person006") return the Person Person006 object
  - id("Person006")/Name returns Name of Person006
- Obviously type safety is guaranteed and used
- If id("Person006") returned an object of type other than PersonType, then id("Person006")/Name would not make sense and give a runtime error

### 13.11. Examples of dotted expressions

- Suppose *e* is a variable for an Emp object and *m* a variable for Manager object. Then
  - *e*.Name is translated as `id(e/ISA/@oID)/Name`
  - *e*.WorksIn.DName is translated as `id(e/WorksIn/@oID)/DName`
  - *m*.DOB is translated as `id(id(m/ISA/@oID)/ISA/@oID)/DOB`
  - The expression *e*.WorksIn.ManagedBy.DOB is translated as  
`e/id(e/id(e/id(e/id(e/WorksIn/@oID)/ManagedBy/@oID)/ISA/@oID)/ISA/@oID)/DOB`
  - The query *what is the date of birth of John's manager* is expressed in OOXQuery, a dialect of XQuery, as follows
    - for *\$e* in doc("PersonnelOODB.xml")/Emp  
where *\$e*.Name = "John"  
return <D> {*\$e*.WorksIn.ManagedBy.DOB} </D>}
  - It would be translated into an XQuery query as follows
    - for *\$e* in doc("PersonnelOODB.xml")/Emp  
where *\$e*/id(e/ISA/@oID)/Name = "John Doe"  
return <D> {*\$e*/id(\$e/id(\$e/id(\$e/id(\$e/WorksIn/@oID)/ManagedBy/@oID)ISA/@oID)ISA/@oID)/DOB} </D>}

## 13.12. How does the translation work?

- Dots force transitions from one object type to another as specified in the schema
- Following transition table is derived from PersonnelOODB.xsd

```
<Map name = "PersonnelOODB">
 <ContextObjectType name = "PersonType">
 <DottedIdentifier name = "@PersonID" transitionCode = "local"/>
 <DottedIdentifier name = "Name" transitionCode = "local"/>
 <DottedIdentifier name = "DOB" transitionCode = "local"/>
 </ContextObjectType>
 <ContextObjectType name = "EmpType">
 <DottedIdentifier name = "@EmpID" transitionCode = "local"/>
 <DottedIdentifier name = "@PersonID" transitionCode = "isa PersonType "/>
 <DottedIdentifier name = "Name" transitionCode = "isa PersonType "/>
 <DottedIdentifier name = "DOB" transitionCode = "isa PersonType "/>
 <DottedIdentifier name = "Salary" transitionCode = "local"/>
 <DottedIdentifier name = "WorksIn" transitionCode = "ref DeptType "/>
 </ContextObjectType>
 <ContextObjectType name = "DeptType">
 <DottedIdentifier name = "@DeptID" transitionCode = "local"/>
 <DottedIdentifier name = "DName" transitionCode = "local"/>
 <DottedIdentifier name = "ManagedBy" transitionCode = "ref ManagerType "/>
 </ContextObjectType>
 <ContextObjectType name = "ManagerType">
 <DottedIdentifier name = "@ManagerID" transitionCode = "local"/>
 <DottedIdentifier name = "@EmpID" transitionCode = "isa EmpType "/>
 <DottedIdentifier name = "@PersonID" transitionCode = "isa EmpType isa PersonType "/>
 <DottedIdentifier name = "Name" transitionCode = "isa EmpType isa PersonType "/>
 <DottedIdentifier name = "DOB" transitionCode = "isa EmpType isa PersonType "/>
 <DottedIdentifier name = "Salary" transitionCode = "isa EmpType "/>
 <DottedIdentifier name = "WorksIn" transitionCode = "isa EmpType ref DeptType "/>
 <DottedIdentifier name = "Budget" transitionCode = "local"/>
 </ContextObjectType>
</Map>
```

- For the context as EmpType, .WorksIn produces transitionCode “ref Dept-Type” which leads to translation id(e/WorksIn/@oID) and the new context is set to DeptType
- The transitionCode for .DOB applied to a ManagerType is "isa EmpType isa PersonType" which leads to translation “ManagedBy/@oID)ISA/@oID)ISA/@oID)/DOB” reflecting two applications of inheritance
- The transition code “local” simply changes “.” to “/”

### 13.13. OOXML and processing technologies for XML

- OOXML a style for object orientation in XML
  - Users can take inheritance, references, and type safety for granted
  - This helps in writing programs that use low level `id(.)` function as well high level dotted expressions, or a mix of the two
  - The nature of programs vary significantly from one processing technology (e.g. DOM, XPath, XQuery, XSLT, etc.) to another
  - Dotted subexpressions have to be converted to applications of `id(.)` so that the program can be executed by the processing technology
- Dotted phrases to `id(.)` conversion
  - It is based upon the transition table mentioned earlier
  - The transition table depends only upon OOXML document schema and independent of the processing technology
  - The table needs to be computed only once for a given scheme and it will work for all XML processing technologies
  - Best place to integrate the dot to `id(.)` conversion is in the compilers of processing technologies
  - In converting `x.A` the determination of the context `x` is highly dependent on the processing technology
- XQuery
  - Structure of queries in XQuery seems much simpler than programs in other processing technologies
  - Instead of a compiler we have implemented a preprocessor that handles most, but not all occurrences of dotted phrases

## Chapter 14

### *Information integration*

- The need for information integration
- Models of integration
  - Multidatabases
  - Data warehouses
  - Mediators
- Decision support
  - Star schema
  - Data cubes
  - Data mining (not covered)
- Acknowledgment
  - Several examples taken from Garcia-Molina, Ullman, Widom book
  - The price of this booklet includes royalty payment to Prentice Hall

## 14.1. Information Integration

- The problem
  - Given two or more databases (called *information sources*)
  - The sources may be conventional databases, possibly of different types, or even files or web pages
  - A user wants to query them as a single database, possibly as a virtual database where information is not stored in materialized form
  - Different users may have different databases  
Or all users may have a central database  
These databases may be *materialized* and/or *virtual*
- Often even sources dealing with the same kind of data differ in subtle ways
  - Therefore, we term source as *heterogeneous* sources
- We consider
  - Federation
  - Warehousing
  - Mediation
- In addition, we consider decision support
  - Multidimensional view of aggregate data
  - OLAP (On Line Analytic Processing)
  - Star schema
  - Data cubes
  - Can be applied for data mining (not covered)



## 14.2. Approaches to Information Integration

- Federated databases.
  - At every (source) site, information is fetched from other sites.
  - There is no central database
- Warehousing.
  - A (*data*) *warehouse* is a centralized physical (materialized) database with its own schema
  - Data in the warehouse is drawn from various data sources
  - Data from each source is transformed to conform it to the warehouse schema
  - The data further processed before it is stored in the warehouse
  - The warehouse is updated periodically, perhaps overnight to monthly
- Mediation.
  - A mediator is centralized virtual database, with its own schema
  - It is a software component, containing no physical data
  - The user may query as if it were materialized
  - The mediator translates the user's query into one or more queries to its sources. The mediator then synthesizes the answer to the user's query from the responses of those sources, and returns the answer to the user.
- Hybrids of above are possible

### 14.3. Running example

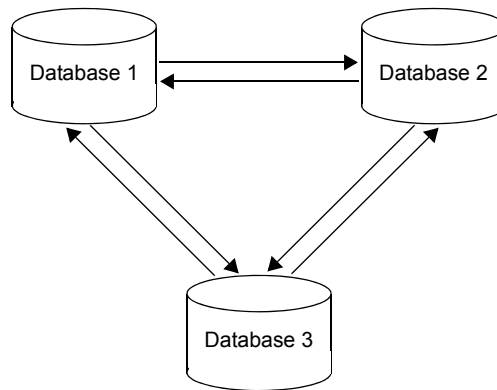
- The Aardvark Automobile Co. has 1000 dealers, each of which maintains a database of their cars in stock.
  - Aardvark wants to create an integrated database containing the information of all 1000 sources.
  - The integrated database will help dealers locate a particular model if they don't have one in stock.
  - It also can be used by corporate analysts to predict the market and adjust production to provide the models most likely to sell.
- For illustration we will often consider two dealers
- Schema for Dealer 1:
  - Cars (serialNo, model, color, autoTrans, cdPlayer, ...)
  - One boolean attribute for every option
  - One relation in the schema
- Schema for Dealer 2:
  - Autos (serial, model, color)
  - Options (serial, option)
  - Two relations in the schema

## 14.4. What schema differences matter to us?

- Recall schema for the two dealers
  - Schema for Dealer 1:  
Cars (serialNo, model, color, autoTrans, cdPlayer, ...)
  - Schema for Dealer 2  
Autos (serial, model, color); Options (serial, option)
- We catalog potential differences among schema
- Different names for similar elements
  - Different relation names: Cars vs. Autos
  - Different attribute: serialNo vs. serial.
- Data type differences
  - Serial number may be string varying length or fixed length or integer
- Different constants for same concept
  - Color may have an integer code or may be a string “black” or “BL”
  - “BL” may be black at one source, but blue at another
- Same constant or names with different semantics
  - A dealer might include trucks in the Cars relation
  - A dealer may or may not distinguish station wagons from minivans
- Missing values
  - A dealer may or may not record color attribute
  - XML database can integrate such data better

## 14.5. Architecture of a federation

- Create one-to-one connections between all pairs of databases that need to share information



- Each connection (arrow) allow a database to query another database that is based on the schema of the latter
- A federated system may be the easiest to build when very limited capability is needed
- For  $n$  databases, potentially  $n(n-1)$  pieces of code are needed.

## 14.6. Example of a federation

- Suppose Dealer 1 wants to query inventory of Dealer 2
  - Dealer 1 composes a list of cars it needs in a relation NeededCars (model, color, autoTrans)

- Recall the two database schemas:

Dealer 1: Cars (serialNo, model, color, autoTrans,...)

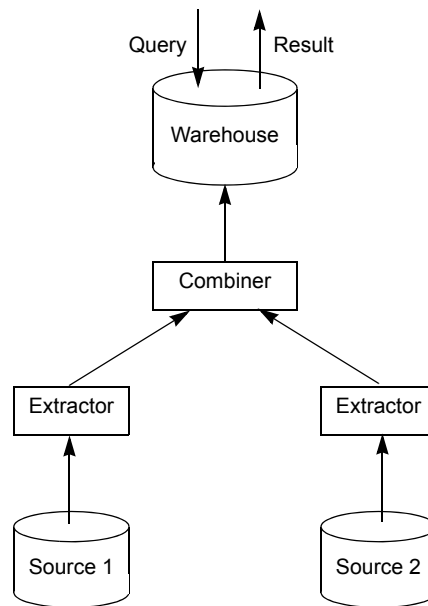
Dealer 2: Autos (serial, model, color); Options (serial, option)

- Dealer 1 executes the following query on Dealer 2's database

```
for each tuple (:m, :c, :a) in NeededCars {
 if (:a = TRUE) { /* automatic transmission wanted */
 select serial
 from Autos, Options
 where Autos.serial = Options.serial
 and Options.option = 'autoTrans'
 and Autos.model = :m
 and Autos.color = :c
 }
 else { /* automatic transmission not wanted */
 select serial
 from Autos
 where Autos.model = :m
 and Autos.color = :c
 and not exists (select *
 from options
 where serial = Autos.serial
 and option = 'autoTrans')
 }
}
```

## 14.7. Architecture of a data warehouse

- Architecture of a data warehouse is shown below



- Information is drawn from multiple sources
- One materialized central database with its own schema
- The warehouse is an ordinary database to a user
- It can be queried as an ordinary database
- On the fly updates to the warehouse generally are forbidden

## 14.8. Populating a data warehouse

- There are at least three approaches to populating a warehouse
- Populating the warehouse periodically
  - This approach is the most common
  - The warehouse is populated once a night, week or month
  - Pros: simple algorithms
  - Cons: Takes long to (re) populate
  - Cons: Data remains out of date for long periods
- Updating the warehouse periodically
  - Incrementally update rather than repopulate the whole database
  - Pros: Faster compared to repopulating an entire warehouse
  - Cons: Algorithms for incremental updates are complex
- Updating the warehouse immediately
  - Each change or a small set of changes at one or more of the sources are propagated to the warehouse
  - Pros: It can support critical applications, e.g., stock trading
  - Cons: Requires more communication and processing

## 14.9. An example of a data warehouse

- The schema of the warehouse

AutosWhse (serialNo, model, color, autoTrans, dealer)

- Note the dealer attribute keeps track of the source

- Recall the two database schemas:

Dealer 1: Cars (serialNo, model, color, autoTrans,...)

Dealer 2: Autos (serial, model, color); Options (serial, option)

- The extractor for the first dealer:

```
insert into AutosWhse (serialNo, model, color, autoTrans, dealer)
 select serialNo, model, color, autoTrans, 'dealer1'
 from Cars
```

- The extractor for the second dealer:

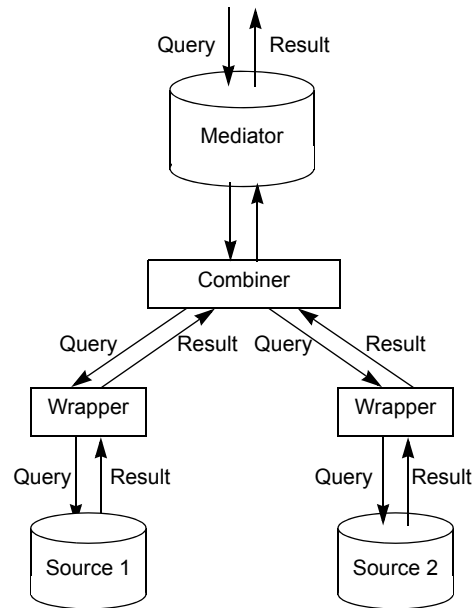
```
insert into AutosWhse (serialNo, model, color, autoTrans, dealer)
 select serial, model, color, 'yes', 'dealer2'
 from Autos, Options
 where Autos.serial = Options.serial
 and option = 'autoTrans'
```

```
insert into AutosWhse (serialNo, model, color, autoTrans, dealer)
 select serial, model, color, 'no', 'dealer2'
 from Autos
 where not exists (select *
 from Options
 where serial = Autos.serial
 and option = 'autoTrans')
```



## 14.10. Architecture of a mediator

- The architecture of a mediator is shown below



- A mediator is a centralized virtual database
- It has a schema but does not physically contain data
- A query is directed toward each source
- A wrapper is needed at each source to transform the query
- Transformed queries are sent to respective sources for execution
- Results are received by wrappers that reformat incoming results
- A combiner combines the results and returns final result to the user

### 14.11. An example of a mediator

- The schema of the mediator

AutosMed (serialNo, model, color, autoTrans, dealer)

- Note the dealer attribute keeps track of the source

- Query to the mediator: *retrieve all red colored cars*

```
select serialNo, model
from AutosMed
where color = 'red'
```

- The mediator transforms the query for each source

- Query for 1st Schema: Cars (serialNo, model, color, autoTrans,...)

```
select serialNo, model
from Cars
where color = 'red'
```

- Query for 2nd schema:

Autos (serial, model, color); Options (serial, option)

```
select serial, model
from Autos
where color = 'red'
```

- The wrapper changes serial to serialNo and returns above to combiner

- Combiner

- Unions two results. Since no serial numbers are repeated there is no need to remove duplicates
- Sometimes a mediator examines the result from one source and may not need to issue queries to other sources. For example consider the query *is there a Camry model in red color*

## 14.12. Wrappers in Mediator-Based Systems

- A mediator is a virtual database with a database schema
  - But a mediator cannot support arbitrary queries
  - The transformation of a query to data sources is difficult to automate
  - More over, all sources may not be databases, some may be flat files and web based
- A template based approach is used
  - A template is an SQL-query allowing variable substitution
  - A collection of template queries for the mediator schema is assembled
  - A template  $T$  is coupled with a translation  $S_i$  for each source  $i$ :  
 $T \Rightarrow S_1, T \Rightarrow S_2, \dots, T \Rightarrow S_n$
  - Translations are associated with the wrappers for the given sources
- When a user query is received by the mediator ...
  - The mediator transforms the query, generates a template query for one or more sources, and sends it to corresponding sources
  - A lookup is performed at each applicable source
  - If all applicable sources have a template for the query, processing is resumed, otherwise it is aborted
  - This is the general idea
  - Mediator and wrappers can be made more intelligent

## 14.13. Example

- Given mediator  
AutosMed (serialNo, model., color, autoTrans, dealer)
- We want to query cars by color.
- Consider the query template for the mediator

```
select *
from AutosMed
where color = '$c'
```
- Dealer 1 schema  
Cars (serialNo, model, color, autoTrans, cdPlayer,...)
- The query for the wrapper for source at Dealer 1 is:

```
select serialNo, model, color, autoTrans, 'dealer1'
from Cars
where color = '$c'
```
- Note that a template query is simple
- Even at that, the number of templates for supporting variable substitutions can be huge
- For our mediator having 5 attributes, there are 32 possible templates
- The number of templates required can be reduced significantly by making wrappers smarter. An obvious capability is filtering

## 14.14. Filter example

- Assume the only template we have is that finds cars given a color.
- To find blue 'Gobi' model cars the query for the mediator is:  

```
select *
from AutosMed
where color = 'blue' and model = 'Gobi'
```
- A possible way to answer the query is to:
  - Use our template to find all the blue cars.
  - Store the result in a temporary relation  
TempAutos (serialNo, model, color, autoTrans, dealer)
  - Then filter out only the model we need:  

```
select *
from TempAutos
where model = 'Gobi'
```
- Such filtering could have been done by each wrapper if they are more intelligent

## 14.15. More intelligent wrappers and mediators

- Find dealers and models such that the dealer has two red cars, of the same model, one with and one without an automatic transmission.

```
select A1.model A1.dealer
from AutosMed A1, AutosMed A2
where A1.model = A2.model
 and A1.color = 'red'
 and A2.color = 'red'
 and A1.autoTrans = 'no'
 and A2.autoTrans = 'yes'
 and A1.dealer = A2.dealer
```

- Each wrapper can ignore the condition  $A1.dealer = A2.dealer$

- Wrapper at the source for Dealer 1 first computes the following

RedAutos (serialNo, model, color, autoTrans, dealer)

- By executing

```
select *
from AutosMed
where color = 'red'
```

- Next the wrapper executes

```
select distinct A1.model, A1.dealer
from RedAutos A1, RedAutos A2
where A1.model = A2.model
 and A1.autoTrans = 'no'
 and A2.autoTrans = 'yes'
```

- Question: What if did not care that the cars do not have to come from the same dealer?
  - Very different capability will be needed.

## 14.16. Decision support: On-Line Analytic Processing

- OLTP: on line transaction processing, the usual database system
- OLAP (On-Line Analytic Processing, pronounced “oh-lap”), generally involves queries that use global aggregations on data
- Compared to OLTP, OLAP queries touch a large part of a database
- OLAP queries provide decision-support to an enterprise
- Decisions involve summary information
- We consider two ways for organizing summaries
  - Star schema
  - Data cubes

## 14.17. Star Schema

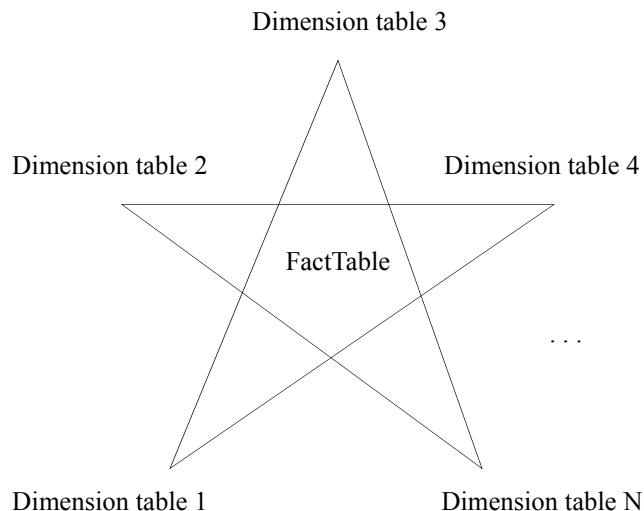
- Suppose the Aardvark Automobile Co. builds a data warehouse to analyze sales of its cars
- The schema for the warehouse might be:

Sales (serialNo, date, dealerName, price)

Autos (serialNo, model, color)

Dealers (dealerName, city, state, phone)

- Sales.serialNo is a foreign key referencing Autos.serialNo
- Sales.dealerName is a foreign key referencing Dealers.dealerName
- Star representation of multidimensional data
  - The star has a fact table at the center (Sales in this case)
  - The dimension tables join with the center (dealer, model, color, city and state in this case)





## 14.18. Star Schema: an example (2 slides)

- Supports queries of the form
  - select grouping attributes and aggregations
  - from fact table joined with zero or more dimension tables
  - where certain attributes are compared with constants
  - group by grouping attributes
- Slicing and dicing are used for decision support
- Example: Suppose the sales of Gobi are below expectation.
  - If we suspect certain colors, we may ask:
    - select color, sum(price)
    - from Sales natural join Autos
    - where model = 'Gobi'
    - group by color
  - This query looks at color *dices* for the *slice* Gobi
- Suppose each color produces similar revenues
  - We might suspect the problem with the color is a recent trend
  - Thus we may try to partition time in to months
    - select color, month, sum(price)
    - from (Sales natural join Autos) join Days on date = day
    - where model = 'Gobi'
    - group by color, month
  - Here, Days is not a conventional stored relation, although we may treat it as if it had the schema
    - Days (day, week, month, year)

## 14.19. Star Schema: an example Slide 2 of 2

- Suppose red Gobis have not sold well recently
  - We may ask if the problem is specific to certain dealers

```
select dealer, month, sum (price)
from (Sales natural join Autos) join Days on date = day
where model = 'Gobi' and color = 'red'
group by month, dealer
```
- Suppose that the sales per month for red Gobis are too small to help observe a trend.
  - Instead of months, partition by years
  - Consider only last two years

```
select dealer, year, sum (price)
from (Sales natural join Autos) join Days on date = day
where model = 'Gobi'
 and color = 'red'
 and (year = 1999 or year = 2000)
group by year, dealer
```

## 14.20. Data cubes: a motivating example

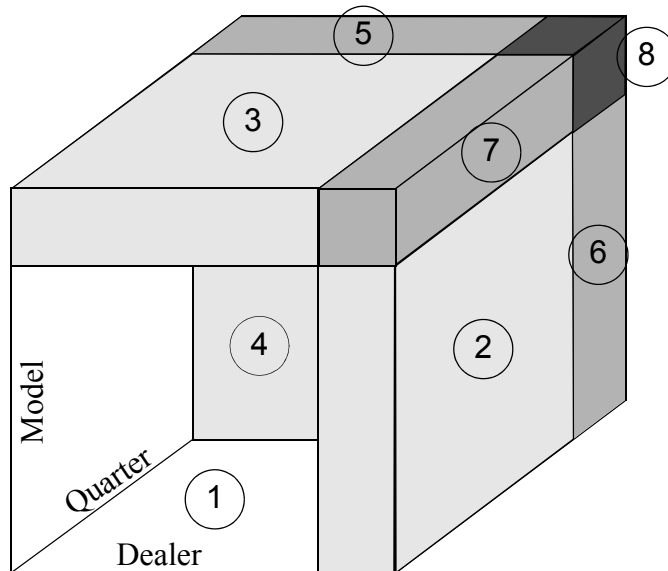
- Data cubes can be used to store and query summary info
- Example: A 2-dimensional cube
  - We want to store and query summary by dealers and models
  - We are given a fact table AutoSales (Dealer, Model, Count, Value)
  - The fact tuple AutoSales( 'Ames Auto', 'Gobi', 10, 200000) says:  
Ames Auto sold 10 Gobi model cars for 200,000 dollars
  - Facts, 1- and 2-dimensional summaries are stored in the data cube

↑ Model	('AA', *, 40, 1500000)	('DW', *, 40, 1360000)	(* , *, 80, 2860000)
Tobi (T)	('AA', 'T', 10, 500000) ('DW', 'T', 10, 600000)		(* , 'T', 20, 1100000)
Robi (R)	('AA', 'R', 20, 800000) ('DW', 'R', 10, 400000)		(* , 'R', 30, 1200000)
Gobi (G)	('AA', 'G', 10, 200000) ('DW', 'G', 20, 360000)		(* , 'G', 30, 560000)
	Ames Auto (AA)	DSM Wheels (DW)	Dealer →

- There are 6 fact tuples in AutoSales
- There are 3 summary tuples of the dealer-dimension  
An example is AutoSales(\*, 'Gobi', 30, 560000)  
It states that all dealers sold 30 Gobies for \$560,000
- Similarly, there are 2 summary tuples of the model-dimension
- AutoSales(\*, \*, 80, 2860000) is the only 2-dimensional summary  
It says 80 cars for \$2,860,000 of all models were sold by all dealers
- Number of tuples
  - Number of fact tuples  $2 * 3 = 6$
  - Count of all tuples in the data cube  $= (2 + 1) * (3 + 1) = 12$
  - Count of summary tuples  $= 12 - 6 = 6$

## 14.21. Data cubes: a 3-dimensional data cube

- Include a dimension for quarters for storage and query  
AutoSales (Dealer, Model, Quarter, Count, Price)
  - Assume that there are 2 dealers, 3 models, and 4 quarters



- Different parts of the cube are explained below
  - Part 1. The fact table consists of  $2 * 3 * 4 = 24$  fact tuples
  - Part 2. AutoSales(\*, M, Q, C, A)  
For each Model M and Quarter Q, C autos have been sold for A \$s  
There are  $3 * 4 = 12$  such summaries
  - Part 3. AutoSales (Dealer, \*, Quarter C, A) contains  $2 * 4 = 8$  summaries
  - Part 4. AutoSales (Dealer, Model, \* C, A) contains  $2 * 3 = 6$  summaries
  - Part 5. AutoSales (Dealer, \*, \* C, A) contains 2 summaries
  - Part 6. AutoSales(\*, Model, \* C, A) contains 3 summaries
  - Part 7. AutoSales(\*, \*, Quarter C, A) contains 4 summaries
  - Part 8. AutoSales(\*, \*, \* C, A) contains 1 summary

## 14.22. Data cubes: syntax for summaries

- Consider AutoSales (model, color, date, dealer, count, value)
  - This can be viewed as a 4-dimensional cube
  - The dimensions are model, color, date, and dealer
- Referring to interesting subsets of tuples of the cube
  - AutoSales('Gobi', 'red', '1999-05-21', 'Friendly Fred', 2, 45000)  
Says: on May 21, 1999, Friendly Fred sold 2 red Gobis for a total of \$45,000
  - AutoSales('Gobi', \*, '1999-05-21', 'Friendly Fred', 7, 152000)  
Says: on May 21, 1999, Friendly Fred sold 7 Gobis of all colors, for a total price of \$152,000.
  - AutoSales('Gobi', \*, '1999-05-21', \*, 100, 2348000)  
Says: on May 21, 1999, there were 100 Gobis of all colors sold by all the dealers at a total price of \$2,348,000.
  - AutoSales('Gobi', \*, \*, \*, 58000, 1339800000)  
Says that over all time, dealers, and colors, 58,000 Gobis have been sold for a total price of \$1,339,800,000.
  - AutoSales(\*, \*, \*, \*, 198000, 3521727000)  
Says: the total sales of all Aardvark models in all colors, over all time at all dealers is 198,000 cars for a total price of \$3,521,727,000.

## 14.23. Data cubes: querying data cubes

- Consider the following relation and query
  - AutoSales (model, color, date, dealer, count, value)
  - ```
select color, avg (value)
from AutoSales
where model = 'Gobi'
group by color
```
 - Summary in date and dealer dimensions is implicit in the query
 - Grouping is by color
 - The scope is limited to Gobi model cars
 - Such query has to process aggregations of large numbers of tuples
- Consider AutoSales ('Gobi', c, *, *, n, v)
 - It readily contains much of desired info
 - Here c, an independent variable, is assigned different color-values
 - Values of dependent variables n and v are extracted from the cube
 - The tuples contain precomputed aggregates in the desired dimensions
- Thus one could ask

```
select c, v/n
from AutoSales ('Gobi', c, *, *, n, v)
```

 - A far more efficient query
- Navigate in the cube looking for answers
 - Drill down for finer details: use fewer *s
 - Roll-up for coarser details: use more *s

Chapter 15

Sorting

In this chapter we will consider sorting a physical file, simply referred to as a file. As usual, we will assume that the file is very large and it cannot fit in the main memory. Using a small number of buffers, a few pages of the file can be accessed at a time for internal processing required for sorting. This type of sorting is called *external sorting*. External sorting is different from *internal sorting* where the whole file can be stored in the main memory at once.

In order to fix notation, we will also assume that sorting will be done in non-decreasing order (\leq) of key values. For the sake of simplicity, in this chapter we will implicitly assume that all pages in the file are full, that is, the space utilization is 100%. Note that a key may consist of several attributes, and it may be a unique or a nonunique key. Throughout this chapter we will only consider the discrete model of the disk, but some exercises will consider cascades. We assume that the file contains N pages and B buffers are available to sort it.

To understand external sorting, we need to introduce the concept of a run. A *run* is a sorted segment of the file. Consider the following example.

Example 15.1. Consider the emp file on Page 56. That file is sorted by Name, but not sorted by ID. Suppose we use ID as our key. Then the sequence of records in the file is $\overline{151}, \overline{306}, \overline{200}, \overline{310}, \overline{101}, \overline{311}, \overline{312}, \overline{305}, \overline{165}, \overline{160}, \overline{300}, \overline{180}$. The sequence consists of the following 7 runs (separated by semicolons): $\overline{151}, \overline{306}; \overline{200}, \overline{310}; \overline{101}, \overline{311}, \overline{312}; \overline{305}, \overline{165}; \overline{160}, \overline{300}; \overline{180}$.

We face a problem that we may not be aware of the runs that are contained in a given file. For example, we may be given a file that happens to be completely sorted. If we are not aware of this, even to realize that the file is sorted, the least we have to do is read all pages in the file once, costing us N page accesses. In order to recognize this problem, we introduce the terms *real run* and *certified run*. A real run refers to a run that is present in a file, irrespective of whether we are aware of it or not. On the other hand, a *certified run* is a sorted segment that we know is a run. When we are given a file without any information about distribution of keys, we may assume that keys are completely out of order, that is, every record forms a run. Thus, the number of certified runs is same as the number of records in the file. The file in Example 15.1. contains 7 real runs and 12 certified runs. The goal of sorting is to reduce the number of certified runs to 1.

We will first present the basis underlying method of external sorting. Although, the cost of processing could be substantial, we do not consider it on a formal basis

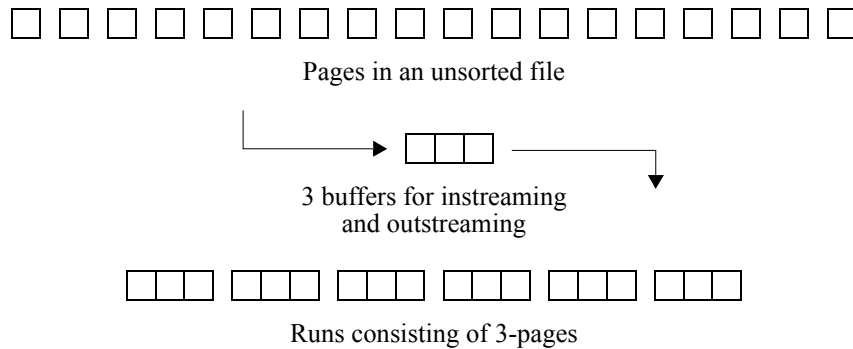


Figure 15.1. Creation of runs for the file of Example

in this book. After introducing the basic method of sorting, we present some interesting ideas that also take the cost of processing into account in order to ask how we could minimize the total elapsed time in sorting a file.

15.1. Basic underlying method of sorting

Essentially, external sorting is done in two phases: *creation of runs*, followed by *merging*. In the phase for creation of runs, segments of the file are brought into the main memory and sorted using an internal sort. At the end of this phase, each segment becomes a run. In the merge phase, the runs are continually merged together to form larger runs until the whole file becomes a single run. The two phases are illustrated through the example given below.

Example 15.2. Consider a file containing 1,800 records. Assume that 100 records fit in 1 page. Therefore, the file contains $1,800/100 = 18$ pages. Assume that $B = 3$, that is, we are given 3 buffers to sort this file.

15.1.1. Phase 1: Creation of runs

For creation of runs, it is appropriate to use all the B buffers that are available to us. These buffers are used to read a segment of B pages from the file, sort them using an internal sort, thereby making and certifying it as a single run, and write the segment back to the file.

For sorting the file in Example 15.2., we are given $B = 3$ buffers. The buffers are used for instreaming and outstreaming. We fill-up the buffers by reading first 3 pages from the given file, sort the contents of the 3 buffers – thereby forming a run a run, and write these three pages to the disk. In terms of cost, 3 pages are accessed, 3 pages are processed, and 3 pages are written. As we are mainly taking the cost of disk accesses in to account, the cost is 6 page accesses. This process is repeated for the next 3 pages of the unsorted file and creating another run consisting of 3 pages,

also costing 6 page accesses. This process is performed a total of 6 times costing $6 * 6 = 36$ page accesses. The result consists of 6 runs of 3 pages each.

15.1.2. The concept of a pass

There is an interesting and simple way of quantifying the cost of creation of runs. Although reading and writing of pages alternate several times, quantitatively we see that eventually all pages are read once as well as written once. The latter, reading all pages once and writing all pages once, is called a pass. We could say that the cost of creation of runs consists of 1 pass. The cost of a pass for a file with N pages is obviously $2N$ page accesses, $2 * 18 = 36$ in this case. Expressing the cost in terms of number of runs is intuitively useful. The file is not yet sorted and we could say that more runs are needed to complete the sorting.

The question also arises about where is the output written. If we do not care about disturbing the original file, we can write the 3-page segment in its original location. In other words, the old 3-page segment gets overwritten as a 3-page run. This procedure is repeated for the remaining 5 segments. The result is that the original unsorted pages of the file get replaced by ones consisting of runs. Although while creating runs, writing pages of a file in their original location is simple and helps save disk accesses, in general this can complicate algorithms and sometimes not even possible. However in databases algorithms are often stream based. Streaming allows easy identification of pages that become dispensable and such pages can be deallocated. We note that instreaming from one disk and outstreaming to a different disk can be considerably faster. If the pages are allocated and deallocated in physical sequence of the disk this is even more helpful. In fact it is best if some space on disk(s) are set aside for sorting until the sorting is completed.

Sorting is a utility. It is rarely used as a structure of choice if the file changes dynamically. It can however be used to bulk load a file to make it the sequence set of a B+ tree. As we will see, a B+ tree helps keep a file sorted even when updates are made. Sorting is also invoked as a utility to speed query processing during certain circumstances when joins are involved.

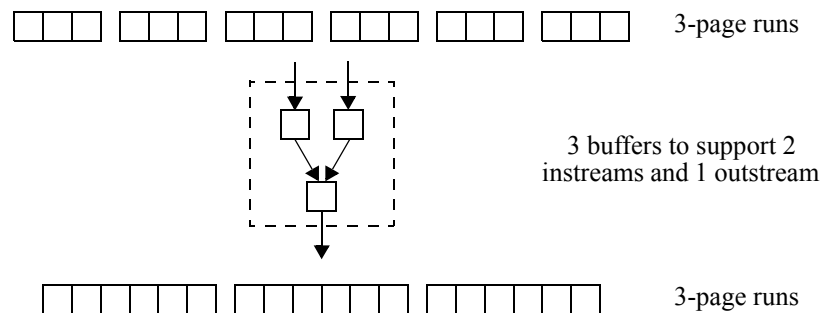
15.1.3. Phase 2: Merging

The main idea in merging is that two or more runs can be merged together to produce one larger run. This processes can be repeated on the given runs until one run is left, at which point the file is sorted.

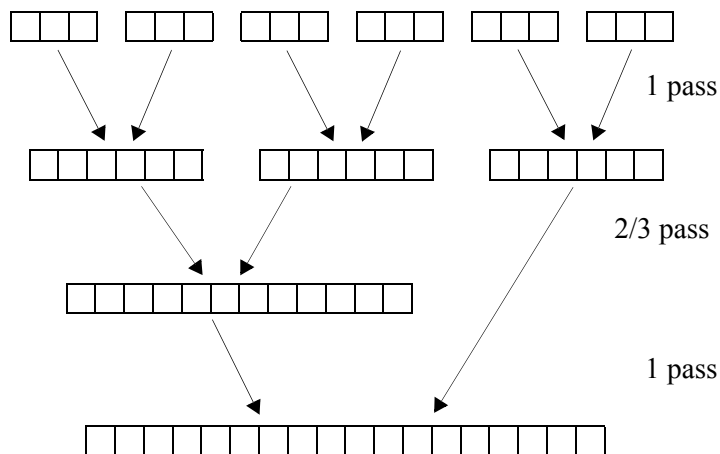
For our running example, let's use the 3 buffers to support two input streams and one output stream as shown in Figure 15.2(a). With this configuration we can merge two runs at a time. Merging allows us to transfer one record at a time from input buffers to the output buffer. The output buffer is written to create a larger run out of the given runs. Whenever an input buffer becomes empty, processing is sus-

pending and the next page from the corresponding run is read into the buffer. Whenever the output buffer becomes full, the processing is suspended and the buffer is written to the disk. The sequence in which the 3 buffers get read and written is unpredictable. (Also, think what happens when a run coupled to one input stream finishes much before the other input stream?)

Merging may require several full and partial passes. For our file, the 2-way merge consists of a binary tree as shown in Figure 15.2(b). In the first pass we reduce 6 runs of 3 pages each into 3 runs of 6 pages each. The second pass is a partial pass; only 2 of the 3 runs can be paired and merged leading to a 12-page run. Now we are left with a 12-page run and a 6-page run. The third pass is a full pass, requiring us to go through the entire file. In this pass the remaining two runs are



(a) Merging of runs for the file of Example 15.2.



(b) The merge tree costing Total of 2 and 2/3 passes

Figure 15.2. Merging

merged together to form a single 18-page run, and the file is now sorted. The complete merging requires 2.67 passes. Including the 1 pass for creation of runs, the total cost of sorting is 3.67 passes. A total of $3.67 \times 2 \times 18 = 132$ pages are accessed.

15.1.4. m-way merging

As stated above, the input-output cost of merging only depends upon the number of passes. Therefore, a merge tree with a small height is favored, and so it is best to try to use m-way merging for the largest possible value of m. However, the cost of merging on to one page increases with the value of m. This is because for larger value of m, one needs to make more comparisons to choose the smallest key from m input streams. This increase in cost is duly offset by the fact that the number of passes reduce.

Example 15.3. In the previous example, we considered 2-way merging. Now let us assume that we have up to 4 buffers available for merging. As shown in Figure 15.3, we can have a pass of 3-way merging followed by a pass of 2-way merging. Merging can be accomplished in 2 passes, as opposed to 2.67 passes for 2-way merging. Even though the processing cost of a pass is greater for 3-way merging than that of 2-way merging, it is interesting to verify that the overall cost of processing is lower due to reduction in number of passes.

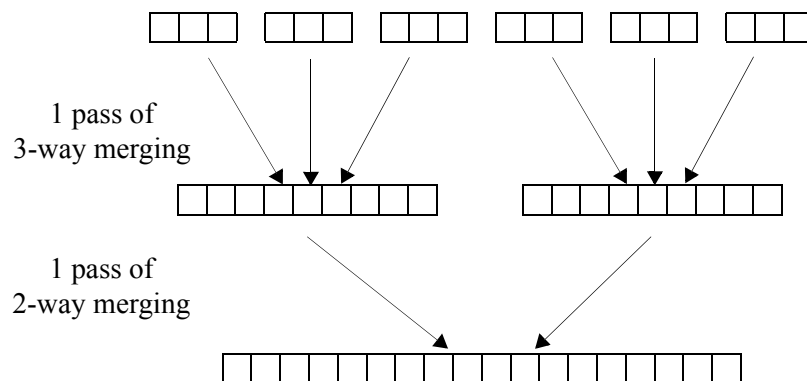


Figure 15.3. Merge tree for Example 15.3.

So far we have introduced a basic elements for sorting a file. Considerable improvements can be made in performance of sorting by using some interesting ideas. These ideas are mentioned below, and explained in detail later.

- **Creating large runs.** We have already seen that with B buffers, we can create runs that consist of B pages. By using a clever technique, we can create runs that are twice as long on average. Although, this does not reduce the cost of creating runs, but it yields fewer runs. The reduced number of runs helps in reducing the cost of merging because fewer passes are needed.
- **Fast m-way merging.** The second improvement is to make m -way merging faster. Ordinarily, choosing the smallest number in m input streams requires us to make $m-1$ comparisons. This can be reduced to $\log_2 m$ comparisons using a clever data structure called tree of losers. This would save processing time that is of critical importance in sorting. It allows one to do m -way merging with a larger value of m . That reduces the number of passes needed during the merging phase.
- **Pipeline merging.** The third improvement is pipeline execution. Here, input, processing, and output are all done in parallel. Although, this idea by itself does not save the processing time or the number of page accesses, it does reduce the time between start and end of sorting by keeping the system dedicated and fully utilized to the task of sorting.

15.2. Creating large runs

The technique for creation of runs described in the previous section 15.1.1 is rigid. It pays no attention to the actual runs present in a file. Instead, independently of the state of the file, it reads fixed length segments of records from the file, sorts them and certifies them as runs.

Now we give an algorithm 15.1.1 that implicitly recognizes existing runs and expand them to large ones. Thus in an extreme case when a file is sorted to begin with, we will know this as a fact at the end of very first pass. Moreover, using the B buffers that are available to us, on average the algorithm creates runs of $2(B-2)$ pages, rather than B pages as seen previously. The main idea of the algorithm is that as the records from the buffers are outstreamed to the current run, the space vacated by them can be filled by records from the input stream in order to give them a chance to be incorporated into the current run.

Figure 15.4 shows how buffers are configured. Two buffers are set aside: one to support instreaming of raw data and another to outstream the current run. The remaining $B-2$ are partitioned into two parts H_1 and H_2 . H_1 will hold the records that will eventually be outstreamed to the current run. The remaining records are held in H_2 for the next run. Note that H_1 and H_2 are disjoint and $H_1 \cup H_2$ covers all the records in the $B-2$ buffers. The size of $H_1 \cup H_2$ is fixed at all times. Initially H_2 is empty and then it grows one record at a time until it take over all the $B-2$ buffers. Likewise, initially H_1 covers the $B-2$ buffers, and then it loses one record at a time until it becomes empty.

We need to understand the relative magnitudes of keys in various parts of the

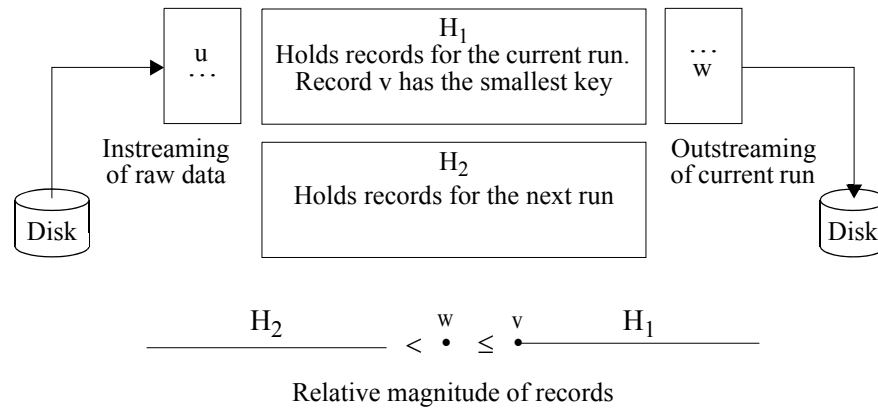


Figure 15.4. Buffer configuration for creation of larger runs

configuration in Figure 15.4. As shown in the figure, suppose the next record waiting to be processed in the input stream is u , the record v in H_1 has the smallest key among all records in H_1 , and the last record that has been outstreamed to the current run is w . Let's denote the keys of u , v , and w as \underline{u} , \underline{v} , and \underline{w} , respectively. Then we can make the following observation.

- Key of every record in $H_2 < \underline{w} \leq \underline{v} \leq$ key of every record in H_1

This is pictorially exhibited in Figure 15.5. The inequalities are to be seen as constraints that should be dynamically satisfied. The magnitude of the key \underline{u} of the record u to be processed next is arbitrary. The following describes the destination of u depending upon the magnitude of \underline{u} relative to \underline{v} and \underline{w} :

- If $\underline{u} < \underline{w}$, then u must wait for the next run. Therefore, it should be directed to H_2 . Recall that $H_1 \cup H_2$ has a fixed size; both H_1 and H_2 are full. This means the smallest record y in H_1 should be outstreamed to the current run, the space vacated by y should be absorbed by H_2 , and this space should be occupied by x . The new smallest y in H_1 should be computed.
- Else if $\underline{u} \leq \underline{v}$, then u must be directly outstreamed to the current run. (What happens if H has become empty and there is no v anymore? We could be in middle of a run.)
- Else, u must be directed to H_1 and wait for its turn to be outstreamed to the current run being processed. This means y should be outstreamed to the current run and the space vacated by y should be occupied by x . The new smallest y in H_1 should be computed.

A close examination of every step should reveal that the constraints mentioned above will be satisfied. It is instructive to observe that at anytime while a sequence of instreaming u -values form a run in the file, no change will occur to H_1 and H_2

and the records u will directly be sent to the output stream.

15.2.1. Implementation

As shown in Figure 15.5, H_1 and H_2 described above can be elegantly implemented as heaps. A heap is a priority queue with the smallest key at the top. It is a binary tree, but efficiently implemented as an array. The $B-2$ buffers can be configured as a single two-headed fixed-size array to store $H_1 \cup H_2$ with tops of H_1 and H_2 at the two ends of the array and the heaps growing inward in the array.

The heaps can support all the operations needed to direct an input record u appropriately. Here are a few examples. The smallest record at the top of H_1 is des-

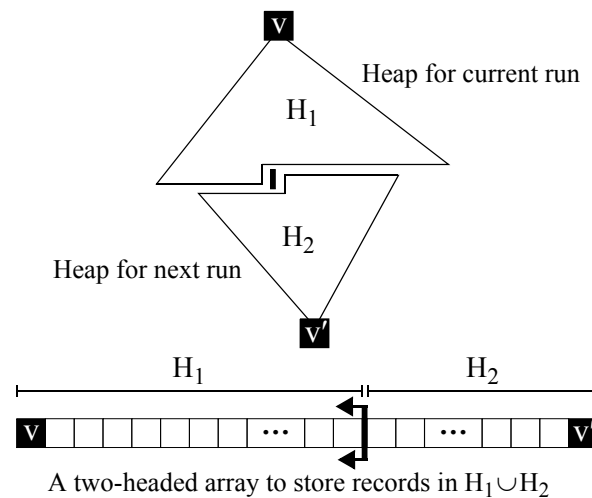


Figure 15.5. Organization of buffers as heaps

ignated as v . Recall that v is removed from H_1 under two different circumstances: v is deleted from H_1 or the deletion of v is followed by an insertion of u . For a heap containing m records, both cases are handled through efficient algorithms for the heaps requiring $\log_2 m$ time. In the former case it leads to removal of the last element of H_1 in the array which is conveniently absorbed as the last element of the heap H_2 . The binary tree containing the records of H_2 is then efficiently redrawn as a heap.

After the current run is completed, the roles of the two heaps are interchanged, and the algorithm can be resumed.

Because of the fact that the H_1 and H_2 occupy only $B-2$ buffers and not B buffers, the technique as described above will create runs that are on average $2(B-2)$ pages long. (This fact is not trivial.) We have already observed that every existing

run in the file will form part of a run being formed.

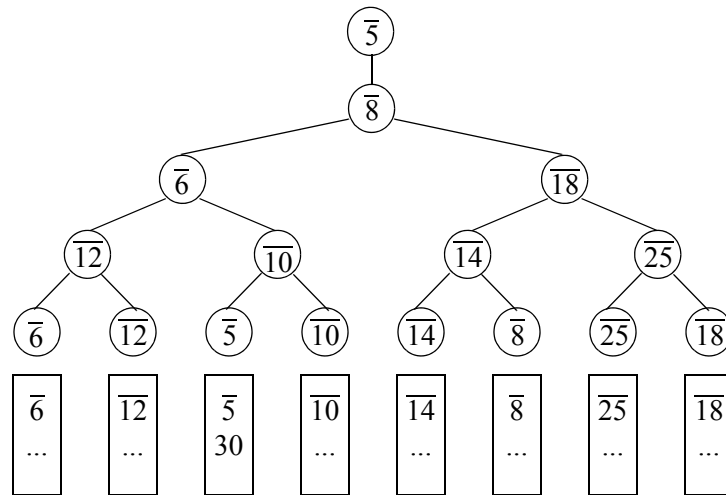
15.3. Fast m-way merging

In m-way merging there are m input streams and one output stream. In such merging m+1 buffers are required, one each for m input streams and one for the output stream. One has to continually determine the smallest of m keys of the records at the front of the input stream and transfer the record to the output stream. Ordinarily, this would require m-1 comparisons. However, while making m-1 comparisons, we not only determine the smallest key, but also come across some information about the relative magnitude among other keys. The *tree of losers* helps us “remember” such information; the next smallest of m keys can be determined in only $\log_2 k$ comparisons rather than m-1 comparisons. The tree of losers is a binary tree. Its leaves are the records waiting to be processed in the m input streams.

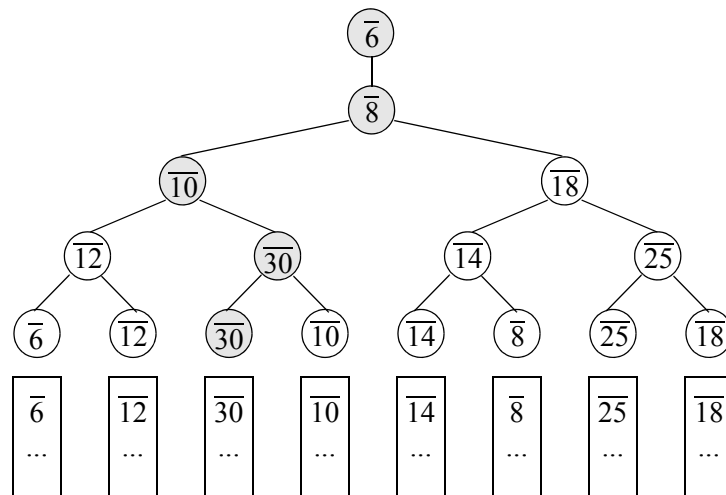
To draw the tree, the keys of pairs of records are compared. We refer to such comparison as a *match* between the two records. The record with the smaller key is said to *win* the match. Figure 15.6(a) shows 8-way merging with tree of losers. The tree corresponds to a tournament with the objective of determining the overall winner. The leaves of the tree are the eight records, $\overline{6}$, $\overline{12}$, $\overline{5}$, $\overline{10}$, $\overline{14}$, $\overline{8}$, $\overline{25}$, and $\overline{18}$, from the front of each input stream. The matches are first played between the four pairs of records. The four losers, $\overline{12}$, $\overline{10}$, $\overline{14}$, and $\overline{25}$, of these matches are installed as parents. Now the matches are played among the four winners $\overline{6}$, $\overline{5}$, $\overline{8}$, and $\overline{18}$. The two losers, $\overline{6}$ and $\overline{18}$, are installed as the parents at the next level. In any step, the match is played between the pairs of remaining losers up to that point. So, the last match is played between the two surviving winners $\overline{5}$ and $\overline{8}$. The loser, $\overline{8}$, is installed as the root of the tree. We are left with the overall winner, $\overline{5}$, that is now installed at the helm of the tree. The helm of the tree is simply a spot in the output buffer.

Now, the tournament must be played again to determine the next winner. For this we do not have to start from scratch. The tournament is started at the leaf level, where the winner originated from, and played only along the path going to the root of the tree. For example, in Figure 15.6 the winner, $\overline{5}$, comes from the third input stream. The next record in that input stream is $\overline{30}$. The tournament will now be among $\overline{30}$, $\overline{10}$, $\overline{6}$, and $\overline{8}$. These are the records that lie on the path from the leaf $\overline{30}$ to the root $\overline{8}$ of the tree. The result of the tournament is shown in Figure 15.6(b). The nodes in the path in question are shaded in Figures 15.6(b). In this tournament the loser is left behind and the winner advances. The overall winner, $\overline{6}$ in this case, is installed at the helm of the tree.

The tree of losers reduces the processing time dramatically. If more memory is available, a larger value of m can be chosen. The memory is needed for the m+1



(a) The first tournament



(b) Subsequent tournaments

Figure 15.6. m-way merging with tree of losers

buffers. A small amount of additional memory is needed for the tree. The tree is a complete binary tree (ignoring the helm) that can be stored as an array of nodes. The child and parent pointers can be computed and need not be stored explicitly in these nodes. Moreover, it is not necessary to store records in these nodes, the pointers to the records suffice. In our example we implicitly assumed that m in m-way

merging is a power of 2. Some thought is needed if the value of m is arbitrary.

15.4. Pipeline merging: minimizing elapsed time

In this section we will consider how merging can be speed up. We will assume that the system permits disk access and processing to be done in parallel. At this point we will pursue an interesting case when a system has two parallel disks and the system is dedicated to our task of merging. We will assume that for m -way merging, m has been chosen such that the time to process one page is same as its access time. Thus we can do merging in a pipeline by performing the following three operations in parallel: read a page, process a page, and write a page. This gives rise to pipeline as shown in Figure 15.7. Therefore, we define a *pipeline operation* to be an operation of the form

$\langle \text{read } \alpha_1, \text{process } \alpha_2, \text{write } \alpha_3 \rangle$

Note that the operation “process α_2 ” refers to merging records from m input buffers to fill one output buffer α_2 . Thus in this operation $m+1$ buffers participate in the processing. A buffer can be involved in only one operation at a time: it can be read, it can be processed, or it can be written. In other words the cost of a pipeline operation is the sum of the costs of its three component operations. However, a pipeline operation allows all three component operations to be performed in parallel, thereby minimizing the elapsed time.

A buffer cannot be involved in I/O and processing at the same time. Therefore, support of uninterrupted m -way merging would require $2k+2$ buffers. The behavior of the output stream is simple, and out of the $2k+2$ buffers, 2 buffers can be permanently coupled to the output stream. The buffer configuration for the input buffers is interesting and will be discussed next.

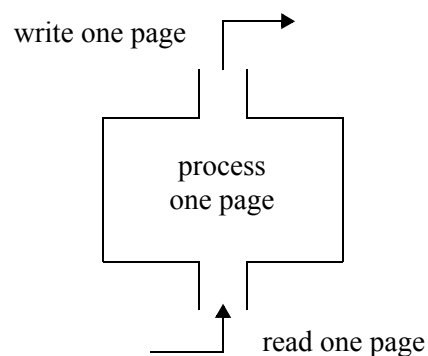


Figure 15.7. Pipeline merging

A simple strategy, termed *static buffer configuration*, is to couple 2 input buffers permanently to each input stream. Example 15.4. shows that the static buffer configuration does not guarantee uninterrupted pipeline.

The alternative is to use a *dynamic buffer configuration*. This configuration maintains the $2k$ input buffers in a pool. A buffer is coupled dynamically to a strategically chosen input stream. It might happen that at a given time some input streams have more buffers coupled with them than others, but it can be shown that the dynamic buffer configuration guarantees uninterrupted pipeline merging.

Example 15.4. This example shows that static buffer configuration does not guarantee uninterrupted pipeline merging. Suppose that for 2-way merging ($m=2$) we are given $2k+2 = 6$ buffers.

Suppose that the 6 buffers are numbered $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \beta_1$, and β_2 . For 2-way merging we need 2 input streams and 1 output stream. As shown in Figure 15.8, we couple buffers α_1 and α_2 to input stream 1, α_3 and α_4 to input stream 2, and β_1 and β_2 to the output stream. A page consists of 2 records. Figure 15.8 also shows 2 runs given to us.

The merging is shown in Figure 15.9. We couple the first run to the first input stream and second to the second input stream. At this point there is nothing in the pipeline. To get the pipeline going, we first read buffers α_1 and α_3 . The result at the end of these operations is shown in Figure 15.9(a). Note that the buffers which are involved in read or write operations are shaded. A shaded buffer is not available for processing. Next we read buffer α_2 and merge into buffer β_1 . Figure 15.9(b) shows the results at the end of this step.

At this point the pipeline is setup: we can simultaneously read a page, we can process a page, and we can write a page. The buffers $\alpha_1, \alpha_3, \alpha_2$, and α_4 are read in a round robin manner. Buffers β_1 and β_2 are likewise written in a round robin manner. The buffer that is not being written is where the processing (merging) is taking place. At this point α_1, α_3 , and α_2 have been read, and β_1 has been processed. Next we attempt to execute the following pipeline operations:

$\langle \text{read } \alpha_4, \text{ process } \beta_2, \text{ write } \beta_1 \rangle$
 $\langle \text{read } \alpha_1, \text{ process } \beta_1, \text{ write } \beta_2 \rangle$
 $\langle \text{read } \alpha_3, \text{ process } \beta_2, \text{ write } \beta_1 \rangle$
 $\langle \text{read } \alpha_2, \text{ process } \beta_1, \text{ write } \beta_2 \rangle$

The configuration after each of the first three pipeline operations is shown in parts (c), (d), and (e) of Figure 15.9, respectively. As shown in Figure 15.9(f), the fourth pipeline operation cannot be executed successfully because we have run out of data prematurely in the buffers for the first input stream, even though there is data left in that input stream on the disk. Therefore, the processing in this example has to be suspended for lack of input.

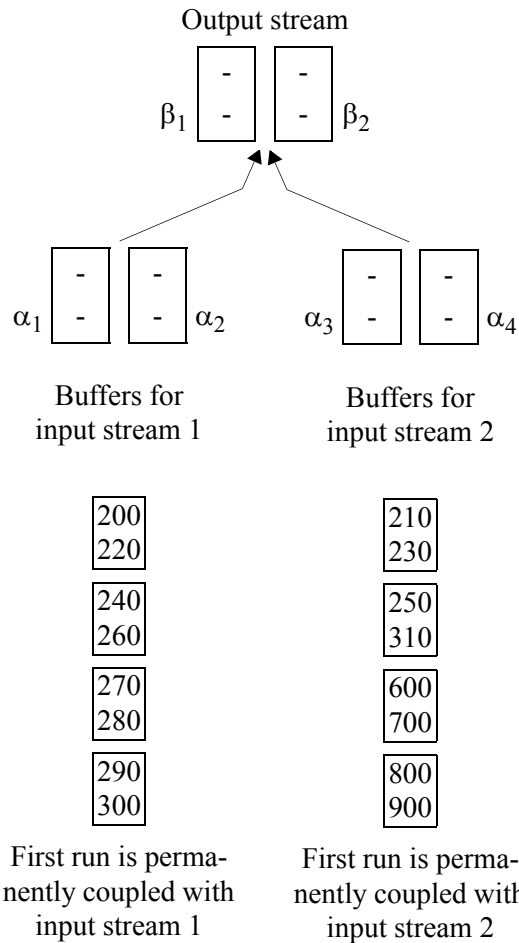
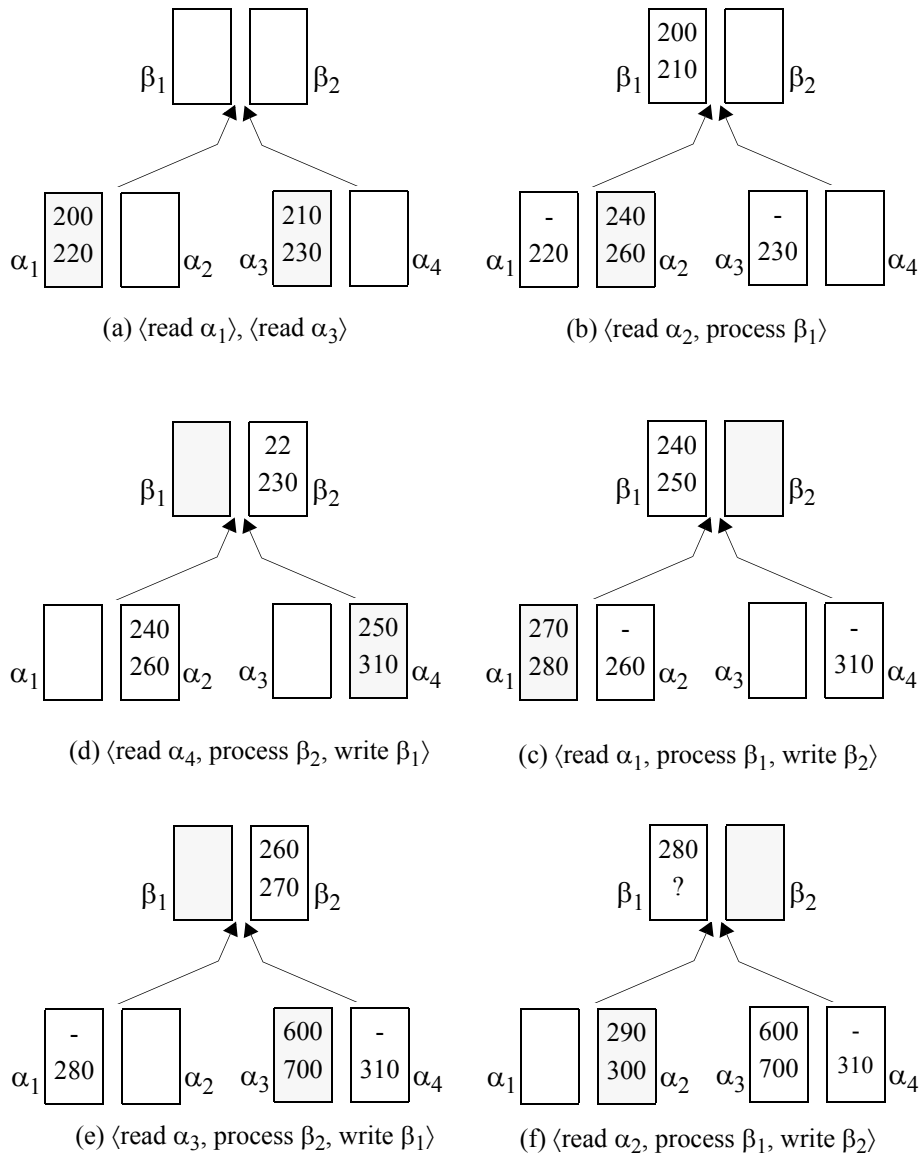


Figure 15.8. Merging setup with static buffer configuration

15.5. Summary

In this chapter we introduced external sorting as a two phase process, creation of runs followed by merging. After having given the basis idea, we discussed how to create larger runs and do faster merging. Then we introduced pipeline merging that allows input, processing, and output to be done in parallel. Although, pipeline merging does not reduce the number of page accesses or the processing time, but helps reduce the elapsed time, i.e., the lapse of time from the start to the end of merging. The processing time in sorting could be substantial and it has to be accounted for. We have not given any analytic models to estimate the cost of sorting a file.



(Shaded buffers are involved in I/O and not available for processing.)

Figure 15.9. Merging with static buffer configuration

We end this chapter with the remark that sometimes the number of page accesses required for sorting N pages with B buffers is informally estimated as $2N \log_{B-1} N$. In practice we can do much better than that.

Chapter 16

Linear hashing

We have assumed that the data is stored on the disk. This means that the retrieval of any information from the disk costs at least one page access. If the information is larger than a page, then the minimum cost is the number of pages occupied by the information. This is an ideal goal and it is not always possible to meet this cost without some additional hidden costs. Hashing is a technique that comes close to achieving the ideal goal. A bit more formally, we are given a key value k , and want to retrieve the record \bar{k} . It should be obvious that in order to achieve the ideal goal the address of the page to reach the record \bar{k} has to be either computed, looked up, or a combination of both purely on the basis of the value k of the key.

In this chapter we start with a basic introduction to hashing. We will mention a couple of classical ideas that cannot be scaled up to large amounts of disk-based large and evolutionary information. In spite of this shortcoming, it is an excellent pedagogical detour and help us understand the issues related to hashing more clearly. The main objective of the chapter is to introduce linear hashing, one of several techniques for hashing that responds well to updates in databases. First we introduce some basic terminology termed key space and address space.

16.1. Key and address spaces

Key space is defined to be the set of all possible keys, and *address space* is defined to be the set of all page (chain) addresses of the file where records are stored.

Example 16.1. Consider a file containing student records at a U.S. university having, 25,000 students. Suppose with a certain space utilization in mind, we expect to store 5 records on average per page. Then we expect about 5,000 pages in the file. At such a university, it is common to use a social security number as a key, let us suppose this is the case. There are a total of 1 billion different social security numbers possible. Therefore in this example the size of the key space and address space are 1,000,000,000 and 5,000, respectively.

To achieve the ideal performance, it seems natural to try to compute the address of the page containing \bar{k} . Because the key value is a clue to the address, the needed address is a function, called a *hash function*, $k \rightarrow H(k)$. Motivated by the above example, we can make a reasonable assumption that the size of the key space is much larger the size of the address space. Under this assumption, a hash function cannot be a one-to-one mapping from key space to the address space. In other

words, a hash function maps more records to a page than the capacity of the page. A *collision* is a condition arising when a new record being inserted hashes to a page that is already full. Clearly, collisions will arise, and we need a procedure, called *collision resolution*, to handle them. Therefore, hashing typically consists of (i) a hash function $k \rightarrow H(k)$, and (ii) a procedure to resolve collisions.

16.2. Classical approach to hashing and its pitfalls

Here we give introduction to hashing covering a couple of interesting examples. Through these examples we will see why these ideas embedded in the examples are not adequate for hashing in databases.

Example 16.2. Consider records whose keys are in the range 101..999. Suppose we have a file with 5 pages with addresses 0..4 such that each page can hold up to 2 records. Records within a page are not necessarily stored in a sorted order. The total capacity of the file is 10 records. Suppose hashing consists of the following:

- **Hash function:** $H_1(k) = k \bmod 5$
- **Collision resolution:** To resolve the collisions, we try page $k+1 \bmod 5$, $k+2 \bmod 5$, $k+3 \bmod 5$, ... until we find a page containing an empty slot.

Suppose we have the following sequence of insertions: $\overline{152}$, $\overline{804}$, $\overline{201}$, $\overline{509}$, $\overline{200}$, $\overline{335}$, $\overline{495}$, and $\overline{399}$.

$152 \bmod 5 = 2$. Hence $\overline{152}$ can be stored in page #2. Similarly, $\overline{804}$ can be stored in #4, $\overline{201}$ can be stored in page #1, $\overline{509}$ in #4, $\overline{200}$ in #0, and $\overline{335}$ also in #0. There are no collisions so far. The state of the file at this point is as shown Figure 16.1(a).

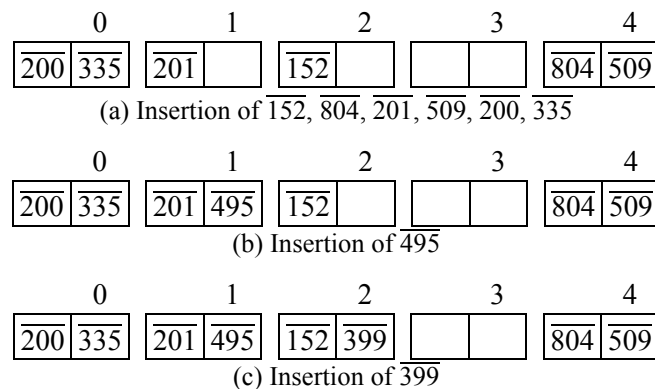


Figure 16.1. Hash file of Example 16.2.

The record $\overline{495}$ hashes to page #0, and thus there is a collision. According to our collision resolution method, we try the next page, which is page #1. Since that page

has an empty slot, we store $\overline{495}$ there. Note that the cost of insertion of $\overline{495}$ is 3 page accesses (read #0, read #1, write #1). The state of the file is now as shown in Figure 16.1(b).

It remains to insert $\overline{399}$, which hashes to page #4. Since page #4 is full, we have a collision. In order to resolve the collision, we try $(399+1) \bmod 5 = \text{page \#0}$. We again have a collision. Next we try page #1 and have yet another collision. Finally we try page #2, find an empty slot, and insert $\overline{399}$ in that page. Note that the cost of this insertion is 5. The final state of the file is now as shown in Figure 16.1(c).

Let us consider the cost of retrieval. What is the cost of retrieval of $\overline{200}$, $\overline{399}$, and $\overline{555}$? Clearly, the cost of retrieving $\overline{200}$ is 1. To retrieve $\overline{399}$, we read the sequence of pages #4, #0, #1, and #2. Therefore, the cost is 4 page accesses. The record $\overline{555}$ is not in the file. However, this is realized only after retrieving the entire file.

Example 16.3. In this example we are considering hashing with chaining. We assume that the records within a chain appear in a sorted order. We use the hash function $H_2(k) = k \bmod 3$. Collisions are simply resolved by chaining additional pages. The result of inserting the records of Example 16.2. is shown in Figure 16.2..

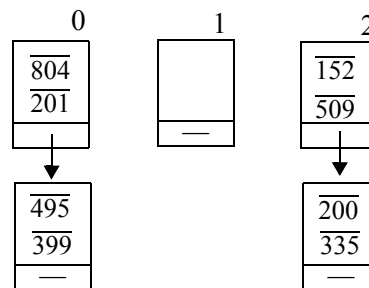


Figure 16.2. Hash file of Example 16.3.

Example 16.2. illustrates a potential problem with hashing: as the space utilization increases, the performance deteriorates. In an extreme (but likely) situation, what if we want to store more than 10 records in the file? One solution is to use chaining. But chaining is an incremental and short-term solution in the life of a growing file. A good solution requires the address space to be expanded, with or without chaining. One way to achieve this is to use a new hashing function with a larger address space and rehash the entire file. But such a solution would require an enormous amount of time for large files.

We need a solution that allows us to enlarge the address space gracefully – affecting a small number of pages. Such a solution is presented in the next section.

The technique is based upon splitting that allows us to expand the address space gracefully. Of course inverse of splitting is collapsing that would also allow a file to shrink gracefully when necessary, e.g. if records are deleted.

16.3. Linear hashing

Linear hashing is a revolutionary technique in hashing. It solves the problems that were pointed out in the previous section. It uses mod function to generate chain addresses. An illustration of hashing using a mod function was given in Example 16.3.. In that example, the address space $\{0,1,2\}$ is fixed. Therefore it has the shortcoming that as a file expands, the chains become longer. Linear hashing provides a way to control chains from growing too large on average. Linear hashing accomplishes this by expanding the address space gracefully, one chain at a time. This graceful expansion is achieved by using splitting.

Suppose we have a file, F , which uses the mod M hash function to determine chain addresses. The file has a static address space consisting of chains with addresses $0, 1, \dots, M-1$. The contents of $[m]$, the chain $\#m$, are as follows:

$$[m] = \{\bar{k} \text{ in } F: k \bmod M = m\}$$

For an example, consider $M = 3$. Then there are three chains $[0]$, $[1]$, and $[2]$. In particular, the chain $[1]$ may consists of

$$[1] = \{\overline{106}, \overline{217}, \overline{151}, \overline{415}, \overline{379}\}.$$

How can we enlarge the address space for this file? To do this we use chain splitting and ask the following three questions.

- How can a chain be split?
- Which chain should be split?
- When should a chain be split?

16.3.1. How can a chain be split?

How do we split a chain $[m]$ evenly into two chains using a mod function? Because we want to expand the address space, the argument for a mod function cannot be to be M as it will only generate existing addresses. However, the answer to the above question is simple: use mod $2M$ to rehash the records in $[m]$. On average, mod $2M$ will hash half of the records to chain $[m]$, and the other half to chain $[M+m]$. This is the main idea behind linear hashing.

Continuing the above example, we see that $2M = 6$. By rehashing the keys in chain $[1]$ we obtain chains $[1]$ and $[4]$. The contents of these two chains are shown below.

$$\begin{aligned} [1] &= \{\overline{217}, \overline{151}, \overline{415}, \overline{379}\} \\ [4] &= \{\overline{106}\} \end{aligned}$$

16.3.2. Which chain should be split?

Figure (a) shows a file where chains arise from mod 3 hash function. To separate new chains to be generated by future splits, a vertical bar, called a *median* is shown. The existing chains are on the left of the median, and the new chains will be added to the right of the median. The median is given the numerical value of 3. Which chain should be designated for splitting? There are the following possibilities:

- split chain [0]: this will create chain [3]
- split chain [1]: this will create chain [4]
- split chain [2]: this will create chain [5]

Linear hashing gets its name from the fact that chains are designated linearly for splitting. In the above situation, we would first split chain [0], next chain [1], and then chain [2]. Note that this is independent of where the insertions are taking place.

As shown in Figure (b), chain [0] is split first. This split first creates a new chain [3]. Next, the records in chain [0] are rehashed using mod 6. The figure also shows the chains covered by mod 3 and mod 6 hash functions. Then, chain [1] is split into chains [1] and [4] (Figure (c)). The third split creates the new chain [5], as shown in Figure (d). At this point we have 6 chains, and all of them are covered by mod 6. This completes one cycle, essentially doubling M from 3 to 6. The new state of the file is shown in Figure (e). Now we can repeat the same procedure as above. During the next cycle, 6 splits will be made and the address space will double to 0..11.

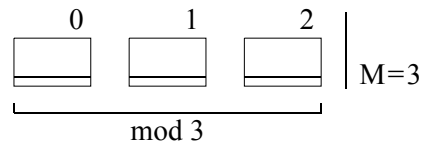
An important feature of linear splitting is that it does not create gaps between existing chains and the new chains. This feature prevents address space from becoming complicated or unnecessarily large. The address space increases gracefully, one chain at a time.

16.3.3. Configuration parameters for a linearly hashed file

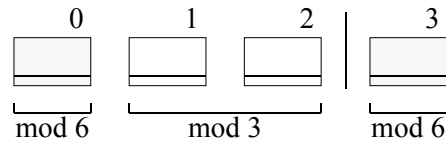
It is easy to characterize the configuration of a linearly hashed file. With parameters sP denoting the chain designated to be split next, and M as the current value of the median, the state of a linearly hashed file is as shown in Figure 16.4.

Example 16.4. To understand the parameters sP and M associated with a linearly hashed file, suppose $M = 6$ and $sP = 2$. This means that in the current cycle of splitting, chains [0] and [1] have been split. Therefore, the state of the file is shown in Figure 16.5.

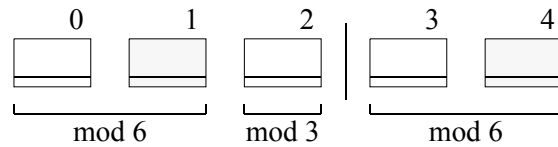
Now consider the record $\overline{319}$. We have $319 \bmod 6 = 1$. Thus we suspect that $\overline{319}$



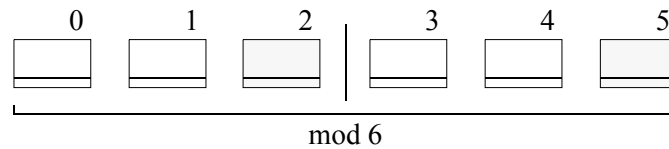
(a) The starting state



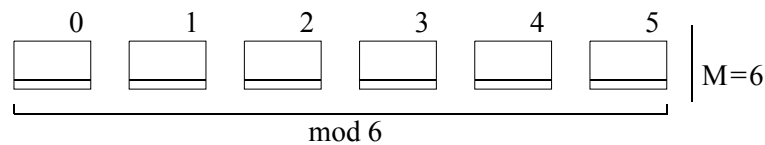
(b) Splitting chain [0]



(c) Splitting chain [1]



(d) Splitting chain [2]



(e) One cycle of splittings completed, M doubles to 6

Figure 16.3. Splitting in linear hashing

is in chain [1]. But chain [1] has been split using mod 12. Thus we must rehash the key: $319 \bmod 12$ gives us 7. This means the record $\overline{319}$ is either in chain [7] or the record is not in the file. As another example consider $\overline{124}$. When we hash it we obtain $124 \bmod 6 = 4$. Because chain number 4 still uses the hash function mod 6, the key 124 need not be rehashed. Therefore we can conclude that the record $\overline{124}$ is either in [4] or the record is not in the file.

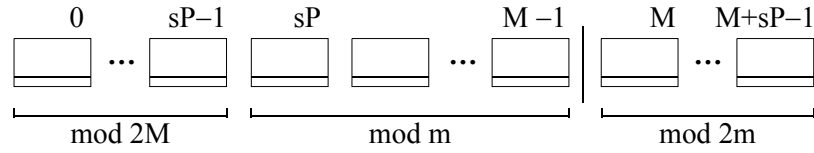


Figure 16.4. State of a linearly hashed file

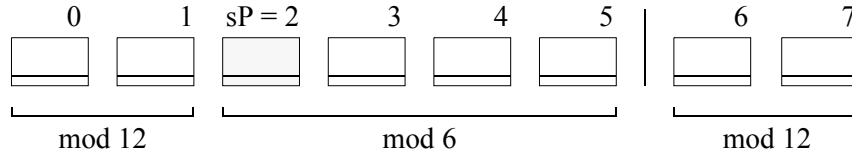


Figure 16.5. State of the linearly hashed file of Example

16.3.4. When should a chain be split?

The chains must be split to keep performance from deteriorating. The basic idea is to ensure that the chains will remain small. Many strategies are possible to accomplish this. In this chapter let's consider average chain length as a trigger for splits. For a given chain, the *chain length* is the number of pages in the chain. The *average chain length* λ_{av} is defined to be the average of all chain lengths in the file. Next we set acceptable lower and upper bounds, λ_l and λ_u , for λ_{av} . In other words, $\lambda_l \leq \lambda_{av} \leq \lambda_u$ are the acceptable limits. A split occurs when λ_{av} exceeds λ_u , and a collapse occurs when λ_{av} drops below λ_l .

16.4. An algorithm for insertion

In context of linear hashing the parameters M and sP help us determine the configuration of a linearly hashed file. These parameters tell us how and which chain to split. We also have introduced the parameters λ_l and λ_u , which trigger splitting. Note that the number of home pages in the file is same as the number of chains, and that number is $M+sP$. Therefore, $\lambda_{av} = N/(M+sP)$, where N is the number of pages in the file. The parameters M , sP , λ_l , λ_u and N are stored in a header of the linearly hashed file. With the help of these parameters, the state of the file can be determined and performance can be controlled. We treat all these parameters as global variables. Before giving the algorithm for insertion of a record in a linearly hashed file, we need to list some auxiliary functions used in the main algorithm.

- LHash hashes a key. This is motivated by Example 16.4.

LHash (k)

{ $m \leftarrow k \bmod M$; if $m < sP$ then $m \leftarrow k \bmod 2M$; return m ;} }

- split(m) splits the chain [m] into chains [m] and [$M+m$].

- $\text{add}(\bar{k}, m)$ adds the record \bar{k} into chain $[m]$. We assume that the chain $[m]$ remains sorted and compacted.

The algorithm LHInsert for insertion of a record \bar{x} into the linearly hashed file is now given in Figure 16.5..

```

LHInsert( $\bar{x}$ );
   $m \leftarrow \text{LHash}(x)$ ;
   $\text{add}(\bar{x}, m)$ 
  while  $\lambda_{av} > \lambda_u$  do
  { split( $sP$ );
    If  $sP < M - 1$ , then  $sP \leftarrow sP + 1$ ;
    else  $\{M \leftarrow 2M; sP \leftarrow 0\}$ ;
  }
end LHInsert;

```

Figure 16.6. LHInsert, an algorithm for insertion

Example 16.5. Now let's consider a comprehensive example to show how this algorithm for insertion works. Assume following:

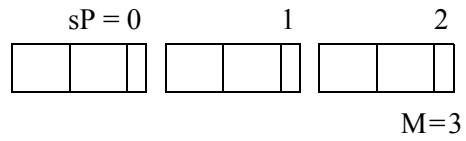
- page capacity of 2 records
- initial value of the median $M = 3$
- lower average chain length $\lambda_l = 1.25$
- upper average chain length $\lambda_u = 1.5$

The average chain length λ_{av} is required to satisfy $1.25 \leq \lambda_{av} \leq 1.5$. We start with an empty file and insert several records one at a time. The starting state of the file is as shown in Figure 16.7(a). Note that $\lambda_l \leq \lambda_{av}$ will be violated when the file is very small. Therefore a certain number of records may be set as a threshold before the requirement $\lambda_l \leq \lambda_{av}$ kicks in. This threshold is not considered by the algorithm in Figure 16.5., and it will be ignored in our example as well.

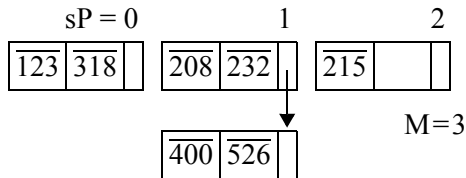
Starting from empty file, we consider insertion of the records $\overline{208}$, $\overline{123}$, $\overline{318}$, $\overline{526}$, $\overline{232}$, $\overline{215}$, $\overline{400}$, $\overline{412}$, $\overline{285}$ and $\overline{345}$ sequentially, one record at a time.

We begin with $\overline{208}$, $\overline{123}$, $\overline{318}$, $\overline{526}$, $\overline{232}$, $\overline{215}$, and $\overline{400}$. These records are inserted in chains whose addresses are determined by applying the hash function mod 3 to their keys. Whenever necessary a new page is chained to absorb the insertion. This is done as long as λ_{av} does not exceed 1.5, which is indeed the case during the insertion of the given records. The state of the file after these insertions is shown in Figure 16.7(b). We note that the average chain lengths are 1, 2, and 1, and their average λ_{av} is $(1+2+1)/3 = 1.33$, which is still in bounds.

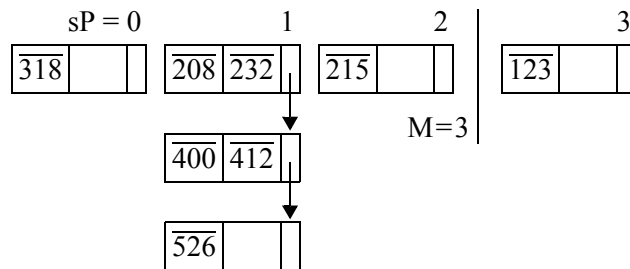
Now we insert $\overline{412}$. As shown in Figure 16.7(c), the record $\overline{412}$ goes in chain $[1]$.



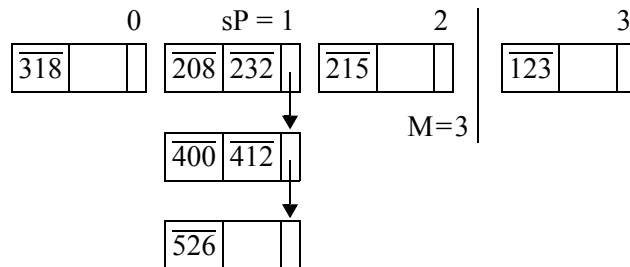
(a) Initial state of the file



(b) First seven insertions, $\lambda_{av} = 1.33$



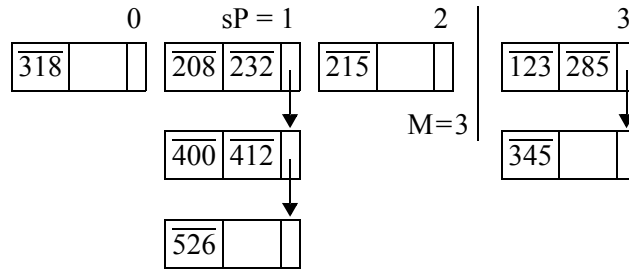
(c) After addition of $\overline{412}$, $\lambda_{av} = 1.67$



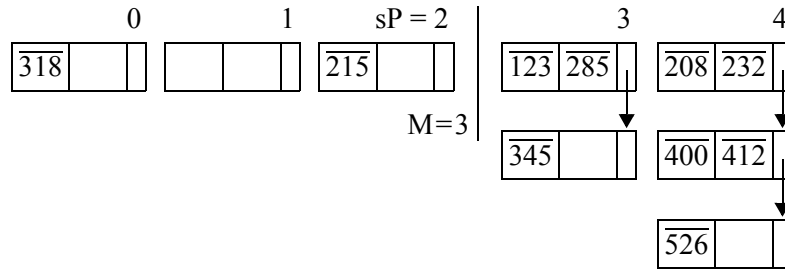
(d) After chain $[0]$ is split, $\lambda_{av} = 1.5$

Figure 16.7. Insertions as in Example 16.5. in a linearly hashed file
(continued on next page)

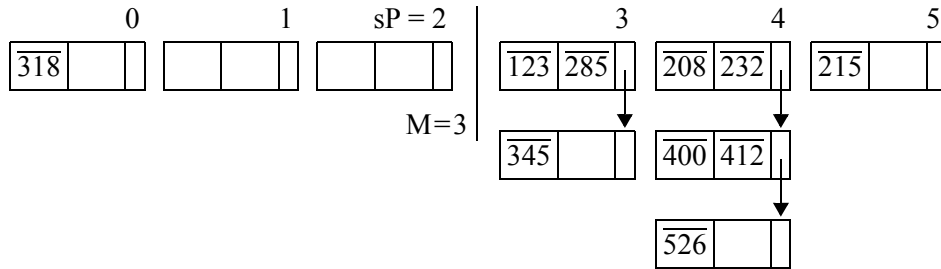
The average length of this chain becomes 3. The average chain length λ_{av} becomes $(1+3+1)/3 = 1.67$, which is not acceptable. Therefore, it is time to split chain $[0]$. The contents of the file after the split are shown in Figure 16.7 (d). After the split we have four chains and $\lambda_{av} = (1+3+1+1)/4 = 1.5$, which is acceptable.



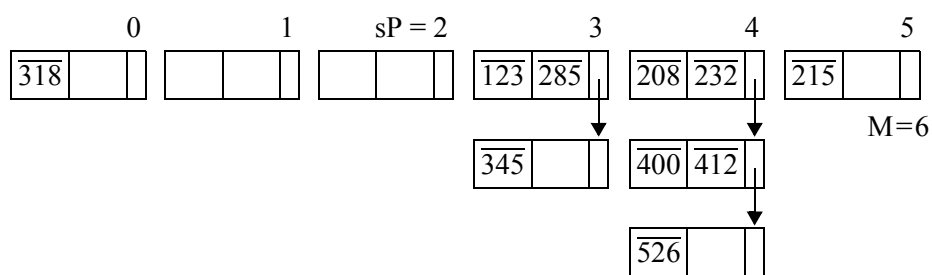
(e) After addition of $\overline{285}$ and $\overline{345}$ $\lambda_{av} = 1.75$



(f) After splitting chain [1], $\lambda_{av} = 1.6$



(g) After splitting chain [2], $\lambda_{av} = 1.5$



(h) sP is reset, M becomes 6, $\lambda_{av} = 1.5$

Figure showing insertions as in Example 16.5. in a linearly hashed file
(continued from previous page)

Now we insert $\overline{285}$ and $\overline{345}$. As in Figure 16.7(e), addition of these keys causes a page to be added to chain [3] violating the condition on λ_{av} . Therefore chain [1] is split. The result after splitting is shown in Figure 16.7(f). The condition for λ_{av} is still violated. Therefore chain [2] is split. The result after splitting chain [2] is shown in Figure 16.7(g). This completes one round of splitting. The parameter sP is reset to 0, and median M is set to 6. λ_{av} is now acceptable. The final state of the file is shown in Figure 16.7(h).

Chapter 17

B-trees

In this chapter we will introduce B-trees. B-trees are perhaps the most mysterious of all file structures. The origin of “B” in “B-tree” is not known. However, “B” does not stand for binary. In fact a B-tree is a search tree where the fanout is at least 3, precisely ruling out the case of a binary search tree. There are several variations of B-trees and these variants are also referred to as B-trees. When a distinction is necessary, B-tree stands for a specific variety that is easiest to describe. Indeed we will first consider this specific variety first. It turns out that in practice this variety is not very useful. So we introduce B+ tree that are highly versatile.

It is useful to put search trees in the context of our discussion. Most readers are probably familiar with binary search trees. A $p+1$ -ary search tree, or a search tree with a fanout of $p+1$, where $p+1 \geq 2$, is a generalization of a binary search tree. In the previous chapter we have seen splitting as a fundamental mechanism to expand a file gracefully. In this chapter we will introduce splitting in search trees. We will see that, from our point of view, the binary search trees ($p+1=2$) are the least interesting of all search trees. The reason for this is that they do not amenable to splitting. Therefore, binary search trees do not readily support graceful dynamic expansion of the address space.

The concept of splitting in a search tree directly gives rise to the definition of a B-tree. A B-tree by definition is a balanced search tree with at least 50% space utilization (except in small degenerate cases). This guarantees a reasonable fanout and a shallow tree, which is good for performance. We will observe that in a B-tree, records are stored in every node, even though the structure of the tree only depends upon the keys. This limits the fanout of a B-tree, making them less than optimal file structures. In the next chapter we will introduce variants of B-trees that are excellent file structures. B-trees are still important to us from a pedagogical point of view. Because of their simple uniform structure they are ideal for gaining a good understanding of operations such as insertions, deletions and retrievals. As the file structures introduced in the next section are simple variants of B-trees, these operations are easily extended to those structures.

17.1. Search trees

A binary search tree is a data structure counterpart of the binary search algorithm. Figure 17.1 shows two of the many possible states of a binary search tree organization of the emp file on Page 56, treating employee ID as the designated key. A node in a binary search tree is of the form (μ_1, \bar{k}, μ_2) ; where \bar{k} is a record

whose key is k , and μ_1 and μ_2 are *pointers*. The pointer μ_1 is called the *left child pointer*, while μ_2 is called *right child pointer*. The keys of all records in the left subtree are $\leq k$ and those of the right are $> k$.

In context of file structures, it is appropriate to assume that a node is a page. Thus following a pointer costs 1 page access. For example, accessing the record 160 requires 5 page accesses in the binary search tree of Figure 17.1(a), and it requires 4 page accesses in the search tree of Figure 17.1(b). A binary search tree is not a good file structure because a node contains only 1 record, and it can have at most 2 child nodes. As a result, the tree may contain long chains, meaning bad performance. It is desirable to store as many pointers in a node as possible, which can help alleviate this performance problem. Therefore let's consider search trees with arbitrary fanout.

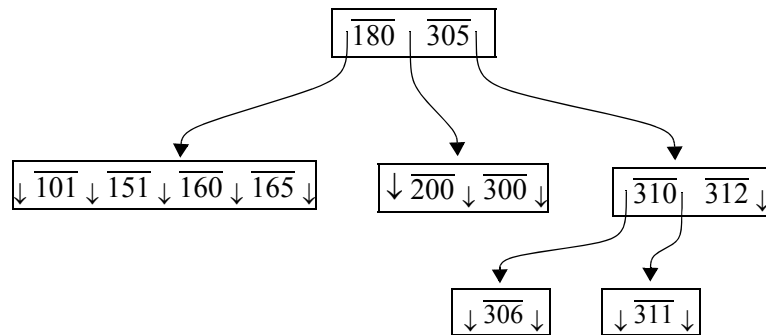
A node in a search tree is termed a *search node*. The records and pointers alternate in a search node: pointer, record, pointer, ..., record, pointer. The number of pointers is always one more than number of records. For a given file we will only consider fixed size nodes (pages) in which we can have up to a fixed number of records and pointers. The number of pointers in a node is called its *fanout*. Clearly, the fanout of a node in binary search tree is 0, 1, or 2. The maximum possible fanout of nodes in a search tree is called the *fanout* of the tree. Thus, the fanout of a binary search tree is 2.

A $p+1$ -ary search tree is a search tree with a fanout of $p+1$. Figure 17.2 shows two of the many possible states of a 5-ary search tree for the same emp file on Page 56. The tree in Figure 17.2(a) contains 6 nodes. Note that all the 6 nodes have the same physical size. Each is capable of holding up to 4 records and 5 pointers. Three nodes contain 2 records and 3 pointers each; not shown in these nodes is the unitized space for 2 records and 2 pointers. Similarly there are 2 nodes containing 1 record and 2 pointers each. Lastly, there is 1 node with 100% space utilization, containing 4 records and 5 pointers.

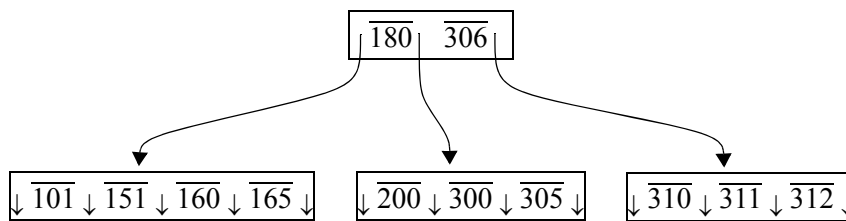
Example 17.1. Suppose pages consisting of 1,024 bytes each are to be used as nodes in a B-tree. Suppose the record size is 80 bytes, and a pointer requires 2 bytes. Then $80 \times p + 2 \times (p+1) \leq 1024$. Optimal value of p is 12. Note that 38 bytes are wasted in every page (wastage amounts to 3.7%).

It should be clear by now that a rule of thumb for a good performance for a $p+1$ -ary search trees it to keep the tree balanced. Formally, a *leaf* is defined to be a node with at least one child pointer that is null. A tree is *balanced* if all its leaves are at the same level. Figure 17.2(b) shows a balanced 5-ary search tree. The binary search tree for the emp file as shown in Figure 17.1(b) is not balanced according to our definition because all leaves of that tree are not at the same level.¹

In addition to having a larger fanout, a 5-ary tree has a critical advantage over



(b) An unbalanced 5-ary search tree



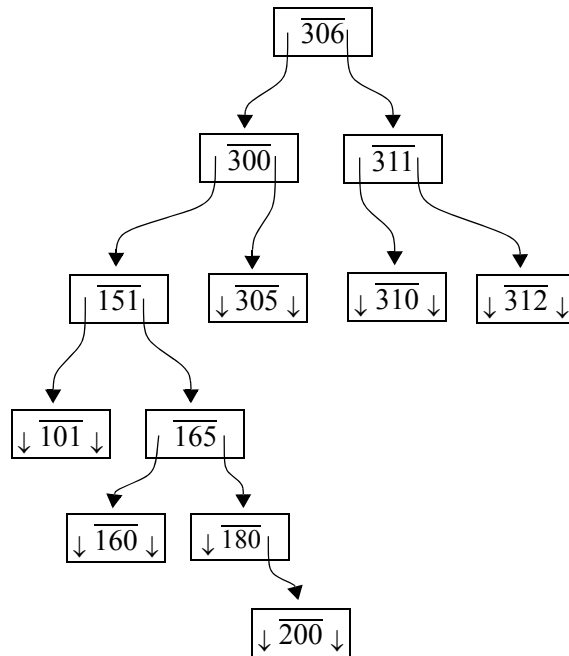
(b) A balanced 5-ary search tree

Figure 17.2. The emp organized as a 5-ary search tree

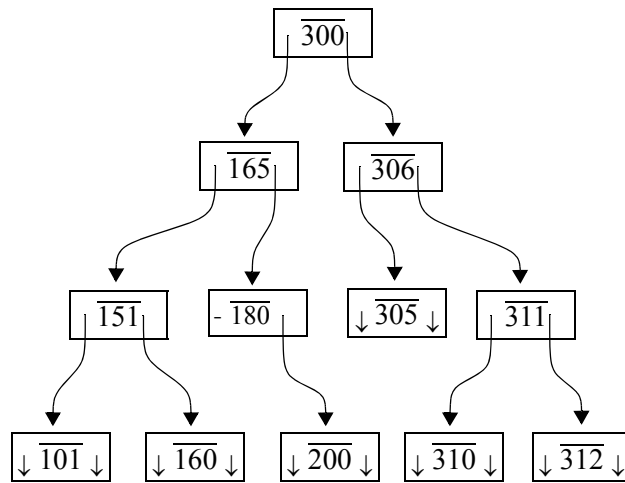
the binary search tree. This advantage is the fact that a 5-ary search node is elastic. Although we have assumed that all nodes in the 5-ary tree have the same physical size, a node can contain anywhere from 1 to 4 records, and consequently 2 to 5 pointers. This choice is not available with a binary search tree. In fact, any $p+1$ -ary tree for $p+1 > 2$ has elastic nodes.

Having defined a balanced tree, we now can focus on how to control the height of the tree. The problem is that even a 100-ary tree can degenerate into a binary search tree, and this is where the concept of splitting comes to the rescue. When a node (page) is split into two nodes, each node still contains about half as many records, thereby preventing a tree from getting degenerating.

1. Note that in data structures texts, a binary search tree is commonly defined to be balanced if all its leaves are at the last two levels. According to that definition, the binary search tree of Figure 17.1(b) would be balanced.



(a) An unbalanced binary search tree



(b) A binary search tree with leaves at the last two levels

Figure 17.1. The emp file organized as a binary search tree

17.2. Splitting in a search tree

As in other file structures, we have some expectations of splitting in a search tree. In every search node, there is a tight relationship between the number of pointers and the number of records: the number of pointers is always one more than the number of records. A split has to break a search node into two search nodes. We have to be careful about the correspondence between numbers of records and pointers in each node. A split should disturb only a small portion of the tree, and the new node obtained after the split has to be integrated into the system at the same level of importance as the original node.

Figure 17.3 shows the split of a 9-ary node. One record is used as a *pivot*, around which the split is performed. In this case the pivot is X_3 . Usually, the middle (or closest to middle) record may be used as a pivot, but this by itself is not a theoretical necessity to carry out a split. When a node is split, a new node μ' is created in addition to the existing node μ . The $\langle \text{record, pointer} \rangle$ pair $\langle X_3, \mu' \rangle$ is added to the parent. Note that in Figure 17.3, the node μ' has been added at the same level as μ . Therefore, if the original tree was balanced, after the split the tree would also be balanced.

17.3. Definition of a B-tree

Before formally defining a B-tree, it is important to state our intention is to split a node in a B-tree only when it overflows. The split is performed around a pivot, which is the middle, or closest to the middle, record. This controls the space utilization in the resulting nodes to a minimum around 50% and prevents the tree from degenerating into a binary search tree. We note that the splits do not destroy the balanced property of a search tree. The splits in a B-tree start at the leaf and migrate upward as needed. In the worst case, a new root is created. We have to allow root this to be an exception: it may contain as few as one record.

A *B-tree* with a fanout $p+1$ is a balanced $p+1$ -ary search tree such that every node, except possibly the root, contains $\lfloor p/2 \rfloor$ records.

Note that a guaranteed space utilization and balanced properties are built into the definition of a B-tree. The feasibility of such a definition follows from the concept of a split introduced in Section 17.2.. Next we will consider the performance of retrievals in B-trees.

17.4. Retrieval performance in a B-tree

In this section we estimate the performance of retrievals in a B-tree. The main consideration will be direct access; i.e., given a key, we want to retrieve the record that has that key. A search ends in a *success* if the record is in the tree, and in *failure* if the record is not in the tree. A node in a B-tree is stored as a page. Every time a

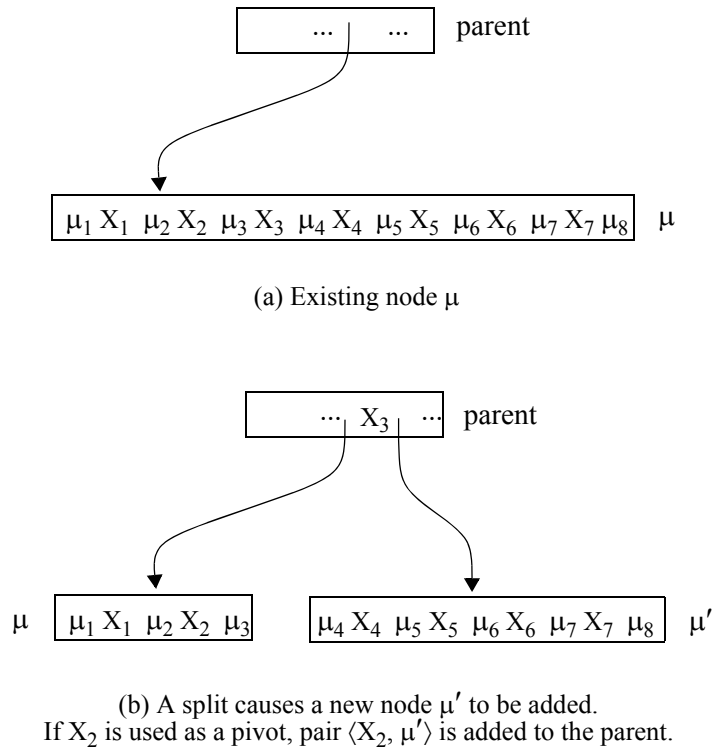


Figure 17.3. Splitting a search node

child pointer is followed, it costs us an additional page access. Therefore the performance of retrievals in a B-tree is directly dependent upon the height of the tree. We want to estimate the height h of the tree in terms of the number of records n and the fanout p of the tree. For brevity, we will denote $\lfloor p/2 \rfloor$ as d .

To facilitate counting, let's imagine that we have an extra level in the tree, which consists of the failure nodes. (See Figure 17.4.). Recall that N denotes the number of nodes (pages) in the tree. The following are the auxiliary parameters:

- h = height of the tree, i.e., the number of levels, counting the root as level 1
- f = the number of (imaginary) failure nodes

The term failure arises from searches that fail because a record we are searching for is not in the file. The parameter f makes it easy for us to give a counting argument. We will calculate f in two different ways in terms of the other parameters and then eliminate f to obtain a relationship among the other parameters.

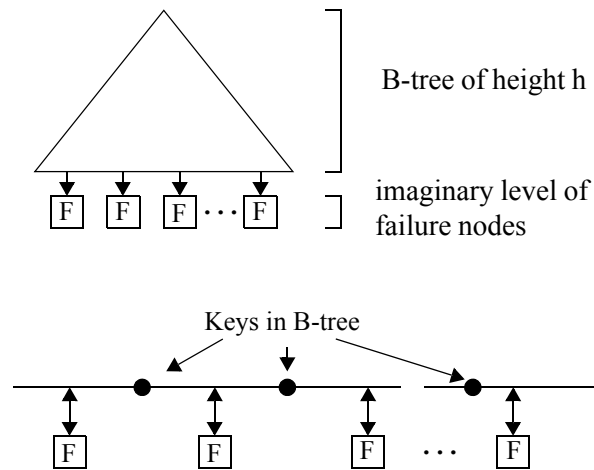


Figure 17.4. Failure nodes

Let's imagine that the set of all possible keys forms the real line. The n keys (records) contained in the tree are points on that real line. If we remove these points, we are left with $n+1$ disjoint line segments on the line. These line segments have an interesting property. Suppose we fix one of the line segments. Then a search of every key in the line segment forces us to traverse a unique path in the search tree leading to the same failure node. Thus, there is a one-to-one correspondence between the line segments and the failure nodes. Therefore,

$$f = n + 1 \quad (1)$$

Now we try estimating f in another way. At level 1, we have at least 1 node. At level 2, we have at least 2 nodes. At level 3, we have at least $2(d+1)$ nodes. At level 4, we have at least $2(d+1)^2$ nodes, and so on. At level $h+1$, we have at least $2(d+1)^{h-1}$ nodes. Thus, we obtain

$$2(d+1)^{h-1} \leq f \quad (2)$$

From (1) and (2) we can eliminate f , giving us the following relationship between d , h , and n :

$$2(d+1)^{h-1} \leq n+1$$

$$\text{Hence } (d+1)^{h-1} \leq \frac{1}{2}(n+1)$$

$$\text{Thus } h - 1 \leq \log_{d+1} \frac{1}{2}(n+1)$$

Finally we have the following estimation for the height of the B-tree in terms of p and n

$$h \leq 1 + \log_{\lfloor p/2 \rfloor + 1} \frac{1}{2}(n+1)$$

Example 17.2. Suppose a file has $n = 1,000,000$ records. We want to store this file as a B-tree with $p = 200$ and we want to estimate the height. Clearly,

$$\begin{aligned}
 h &\leq \log_{\lfloor p/2 \rfloor + 1} \frac{1}{2}(n+1) + 1 \\
 &= 1 + \log_{100+1} \frac{1}{2}(1000000+1) \\
 &= 1 + \log_{101} 500000.50 \\
 &= 1 + \log 500000.50 / \log 101 \\
 &= 3.85
 \end{aligned}$$

Since h is an integer, $h \leq 3$. Directly from the definition of a B-tree, it is easily proved that $h > 2$. Thus $h = 3$.

The above example is significant because it tells us that if we can choose p to be 200, a file containing as many as 1,000,000 records can be stored as a 3-level B-tree. This means that a record with a given key can be accessed within 3 page accesses. In fact the above argument can be strengthened to prove that even 2,000,000 records with the given fanout will require only 3 levels.

It turns out that in a B-tree a node (page) contain records, and it may be difficult to pack 200 records in it. However, the next chapter will introduce a variant of the B-tree called a B^+ -tree. In all but the last level of a B^+ -tree we store keys instead of records, and $p = 200$ is not unrealistic for such a tree. For the above file, we will have a B^+ -tree of height of 4. As the root can be replicated in the main memory requiring only 1 buffer, a performance of 3 page accesses can be achieved. This will be discussed in detail in Example 17.4.. In fact in that example we argue that if we can afford merely 20 buffers to replicate the first one or first two levels in the main memory, we can achieve a guaranteed performance of 2 page accesses.

Now let's switch our attention to insertions and deletions in B-trees. These operations have to maintain all the properties of a B-tree.

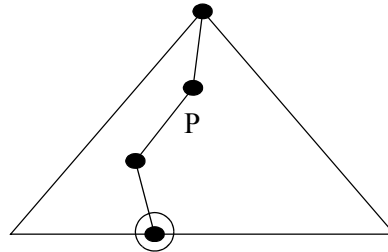
17.5. Insertion in a B-tree

In this section we will discuss BTreeInsert, an algorithm for insertion in a B-tree. We will assume that a B-tree with a pointer, b , to its root is given. We are given a record, \bar{x} , which is to be inserted in the tree.

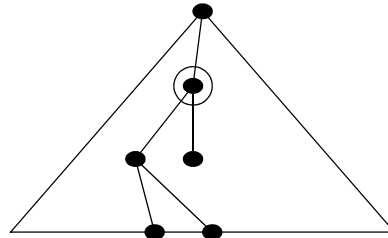
The algorithm for insertion is directly inspired by the concept of split in a search tree introduced earlier. First we traverse down the tree and find a leaf node where \bar{x} logically belongs. See path P in Figure 17.5(a). Next we would like to add the \bar{x} to that leaf node. But we cannot add a record to a node in a search tree without also adding a pointer. Thus addition to a node must consist of a $\langle \text{record}, \text{pointer} \rangle$ pair. Because all pointers in a leaf are null, the record pointer pair to be added to the leaf node in this case is $\langle \bar{x}, \text{null} \rangle$. If the node has space in it to accommodate a pair, the insertion is completed. On the other hand, if the leaf overflows, it is split and a $\langle \text{record}, \text{pointer} \rangle$ pair is added to the parent node. The record in this case is the middle

record of the overflowing node, and the pointer is the address of the newly created node by the split. If the parent node has space in it, the insertion is completed, or else the parent is split, and a $\langle \text{record}, \text{pointer} \rangle$ pair migrates to its parent. This continues until a node that can absorb a record is encountered. See Figure 17.5(b). In the worst case, the root may have to be split, and a new root would be created. See Figure 17.5(c). Note that while splitting we need to revisit the parent nodes backwards along the path P.

(a) Starting from the root the tree is traversed to reach the leaf where the record being inserted logically belongs.



(b) The addition of the record starts at the leaf. If necessary a node is split, and the middle record is moved to the parent. Splitting starts from the leaf and proceeds toward the root until a node is found that is not full.



(c) In the worst case all nodes from the leaf to the root are full. In that case the root is also split and a new root is created.

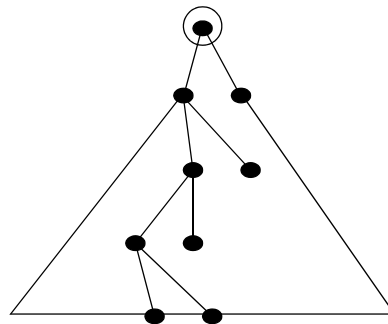


Figure 17.5. Insertion in a B-tree

We assume that we have 1 buffer, α , which allows us to traverse the tree starting from the root to a leaf node. With 1 buffer we do not have the capability of memo-rizing a parent. Therefore, we maintain a stack, denoted as *parentStack*, to store the pointers of all the parent nodes. For example, in the 4-level tree of Figure 17.5, we would push 2 parent pointers on to the stack before the leaf node is reached. The leaf node is not pushed on to the stack. We also assume that there is an additional

buffer, β , so that a split can be composed in the main memory in the 2 buffers, α and β . When a split is performed, 2 child nodes are written, and the parent node is read; the address of the parent is found on the top of the stack. Finally, the parent node has to be modified.

```

BTreelInsert (b,  $\bar{x}$ )
[Insert a new record  $\bar{x}$  in B-tree rooted at b.]

[If the B-tree is empty add the root node with  $\bar{x}$  in it:]
If b is null,
    then  $\{\alpha \leftarrow (\text{null}, \bar{x}, \text{null}); \text{write } \alpha \text{ to a new node } \mu; b \leftarrow \mu\}$ 
[Find leaf  $\mu$  where  $\bar{x}$  belongs. Stack all node addresses from root to just
before  $\mu$  on the parent stack. Leave contents of  $\mu$  in buffer  $\alpha$ .]
FindLeafAndStackParents (b, x, parentStack,  $\mu$ ,  $\alpha$ );
[Setup pair  $\langle \bar{k}, v \rangle$  and buffer  $\alpha$ .]
 $\langle \bar{k}, v \rangle \leftarrow \langle \bar{x}, \text{null} \rangle$ ; [Leaf  $\mu$  is already in buffer  $\alpha$ .]
While  $\alpha$  is full do
    [Compose the split and consolidate on the disk:]
     $(\alpha, \text{midKey}, \beta) \leftarrow \alpha \cup \langle \bar{k}, v \rangle$ ;
    write  $\beta$  to a new node  $\mu'$ ; write  $\alpha$  to  $\mu$ ;
    If  $\mu$  is the root,
        then  $\{\text{write } (\mu, \text{midKey}, \mu') \text{ to a new root; stop;}\}$ 
    [Setup pair  $\langle \bar{k}, v \rangle$  and buffer  $\alpha$ .]
     $\langle \bar{k}, v \rangle \leftarrow \langle \text{midKey}, \mu' \rangle$ ;
     $\mu \leftarrow \text{pop}(\text{parentStack})$ ; read the node  $\mu$  in  $\alpha$ ;
endwhile
[Add the last  $\langle \bar{k}, v \rangle$  to  $\alpha$  and consolidate on the disk:]
 $\alpha \leftarrow \alpha \cup \langle \bar{k}, v \rangle$ ; write  $\alpha$  to  $\mu$ ;
end BTreelInsert;

```

Figure 17.6. The algorithm BTreelInsert

The algorithm BTreelInsert is given in Figure 17.6. The algorithm uses the following procedures.

- **FindLeafAndStackParents (b, k, parentStack, μ , α)** searches for the leaf node where the record \bar{k} logically belongs, stores parent pointers in parentStack, sets μ to point to the leaf node, and reads the contents of μ into the buffer α .
- **$\alpha, \text{midKey}, \beta \leftarrow \alpha \cup \langle \bar{k}, v \rangle$** facilitates a split. It identifies the middle record of $\alpha \cup \langle \bar{k}, v \rangle$ as the pivot, stores all records and pointers in $\alpha \cup \langle \bar{k}, v \rangle$ on the left of the pivot in α , and those on the right of the pivot in β .

Example 17.3. In this example we illustrate insertions in a 5-ary B-tree. For a 5-ary B-tree $p = 4$. Every node, except possibly the root, will have 2 to 4 records and 3 to

5 child pointers. The root is an exception: it can have 1 to 4 records and 2 to 5 child pointers. In a B-tree the child pointers in leaves are nulls. Every child pointer in every non-leaf node is non-null.

We will start with an empty B-tree and insert a sequence of 20 records in the tree. The records to be inserted are $\overline{07}$, $\overline{15}$, $\overline{02}$, $\overline{05}$, $\overline{11}$, $\overline{03}$, $\overline{10}$, $\overline{01}$, $\overline{04}$, $\overline{13}$, $\overline{06}$, $\overline{19}$, $\overline{12}$, $\overline{08}$, $\overline{14}$, $\overline{16}$, $\overline{20}$, $\overline{18}$, $\overline{17}$, $\overline{09}$. Note that we have chosen consecutive records from $\overline{01}$ to $\overline{20}$ for the ease of reading. We start with the empty B-tree b , which is simply a null pointer. A null B-tree is shown in Figure 17.7(a).

The first record to be inserted is $\overline{07}$. Since the tree is empty, the statement $\alpha \leftarrow (\text{null}, \overline{x}, \text{null})$ is executed. It composes a node in the buffer α with the record $\overline{07}$ and two pointers that are null. The buffer α is then written to a page μ on the disk. The pointer b representing the tree is set to μ . Figure 17.7(b) shows the B-tree at this point. The B-tree consists of a single node. This node is the root as well as a leaf. Since this node is the root, it is allowed to contain only 1 record and 2 pointers. The pointers are currently null. Although not shown in the figure, the node contains additional space for three records and three pointers.

Next we insert $\overline{15}$. On entering the algorithm, we form the record pointer pair $\langle \overline{15}, \text{null} \rangle$. This pair is added to the leaf node by collating it with existing record and pointers in the leaf. The records $\overline{02}$ and $\overline{05}$ are inserted in a similar manner. The state of the tree at this point is shown in Figure 17.7(c). The single node in the tree is full: it contains four records and five pointers. All pointers are null.

Now we need to insert $\overline{11}$. On entering the algorithm, we form the record pointer pair $\langle \overline{11}, \text{null} \rangle$. This pair has to be added to the leaf node μ . We know that this is going to split the node μ . In our algorithm a split is expressed as the assignment statement. $(\alpha, \overline{\text{midKey}}, \beta) \leftarrow \alpha \cup \langle \overline{k}, v \rangle$. Here α and β are buffers. The buffer α contains $(\text{null}_1 \overline{02} \text{null}_2 \overline{05} \text{null}_3 \overline{07} \text{null}_4 \overline{15} \text{null}_5)$, the contents of the node μ . The pair $\langle \overline{k}, v \rangle$ is $\langle \overline{11}, \text{null}_{\text{new}} \rangle$. We have added subscripts only for our visual inspection. The environment for the split is now set.

Now we see how the split is carried out by the assignment statement $(\alpha, \overline{\text{midKey}}, \beta) \leftarrow \alpha \cup \langle \overline{k}, v \rangle$. Computation of $\alpha \cup \langle \overline{k}, v \rangle$, the right hand side, leads to a hypothetical node $(\text{null}_1 \overline{02} \text{null}_2 \overline{05} \text{null}_3 \overline{07} \text{null}_4 \overline{11} \text{null}_{\text{new}} \overline{15} \text{null}_5)$. This node does not have to be computed explicitly, but it is only for the ease of understanding a split. This hypothetical node has 5 keys and 6 pointers, and we refer to this as an overflow. Now let us consider the left hand side of the assignment statement $(\alpha, \overline{\text{midKey}}, \beta) \leftarrow (\text{null}_1 \overline{02} \text{null}_2 \overline{05} \text{null}_3 \overline{07} \text{null}_4 \overline{11} \text{null}_{\text{new}} \overline{15} \text{null}_5)$. This is three assignments rolled into one: $\alpha \leftarrow (\text{null}_1 \overline{02} \text{null}_2 \overline{05} \text{null}_3)$, the first half of the overflow node; $\overline{\text{midKey}} \leftarrow \overline{07}$; and $\beta \leftarrow (\text{null}_4 \overline{11} \text{null}_{\text{new}} \overline{15} \text{null}_5)$, the second half of the overflow node. This completes the composition of the split. Note that all splits are composed in the manner described in this paragraph.

It remains to consolidate this split to the disk. First, a new page μ' is allocated

\xrightarrow{b}

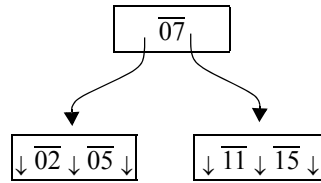
(a) The empty B-tree

\xrightarrow{b} $\boxed{\downarrow \overline{07} \downarrow}$

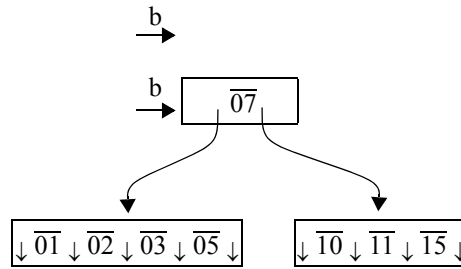
(b) After insertion of $\overline{07}$

\xrightarrow{b} $\boxed{\downarrow \overline{02} \downarrow \overline{05} \downarrow \overline{07} \downarrow \overline{15} \downarrow}$

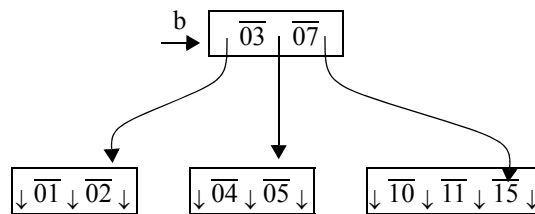
(c) After insertion of $\overline{15}, \overline{02}, \overline{05}$



(d) After insertion of $\overline{11}$

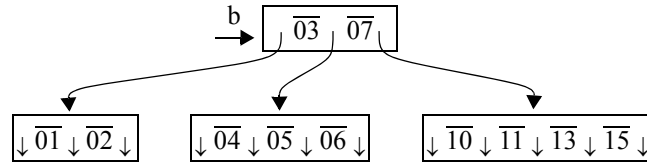


(e) After insertion of $\overline{03}, \overline{10}, \overline{01}$

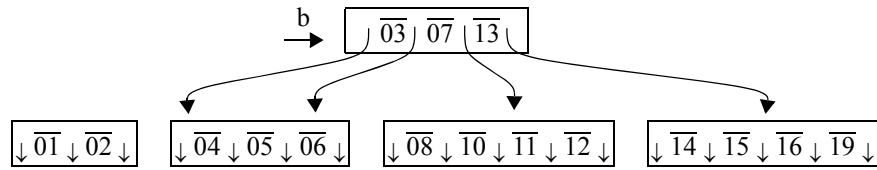


(f) After insertion of $\overline{04}$

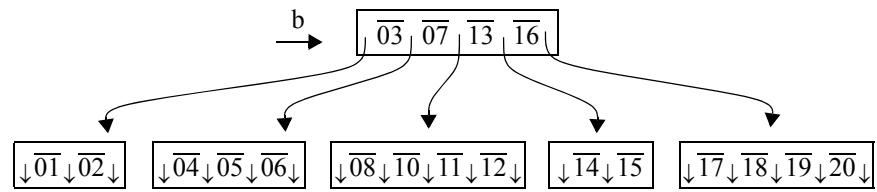
Figure 17.7. Insertions in a B-tree
(Continued on next page)



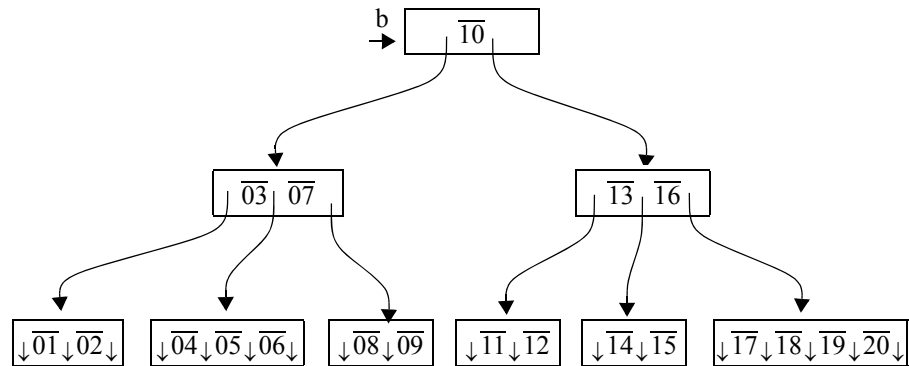
(g) After insertion of $\overline{13}$, $\overline{06}$



(h) After insertion of $\overline{19}$, $\overline{12}$, $\overline{08}$, $\overline{14}$, $\overline{16}$



(i) After insertion of $\overline{20}$, $\overline{18}$, $\overline{17}$



(j) After insertion of $\overline{09}$

Figure continued from previous page

and the contents of β , the second half of the overflow node are written to the disk. Then the original contents of the page μ on the disk are overwritten with β , first half of the overflow node. Usually, the record $\overline{\text{midKey}}$ would be added to the parent. However, in this case there is no parent node because μ is the root b . Therefore, a new page is allocated, $(\mu, \overline{\text{midKey}}, \mu')$ is written to this page, and the page is made the root node by storing its address in b . The state of the B-tree at this point is shown in Figure 17.7(d).

Next, records $\overline{03}$, $\overline{10}$, and $\overline{01}$ are inserted in the B-tree without requiring any restructuring. The B-tree after these three insertions is shown in Figure 17.7(e).

Now the insertion of $\overline{04}$ splits the left leaf in the tree. The middle record $\overline{03}$, together with a pointer to the newly allocated node are added to the parent, the root in this case. The state of the B-tree at this point is shown in Figure 17.7(f).

The insertion of the records $\overline{13}$ and $\overline{06}$ does not cause any further restructuring of the tree. Figure 17.7(g) shows the state of the tree after these insertions.

The insertion of $\overline{19}$, $\overline{12}$, $\overline{08}$, $\overline{14}$, and $\overline{16}$ is similar to those discussed in the previous paragraph. $\overline{19}$ causes the right child of the previous tree to be split. The middle record $\overline{13}$, together with the pointer to the newly allocated node is added to the parent. Figure 17.7(h) shows the state of the tree at this point. Similarly, Figure 17.7(i) shows the state of the tree after addition of $\overline{20}$, $\overline{18}$, and $\overline{17}$.

Now we consider the insertion of $\overline{09}$. The addition of the record pointer pair $\langle \overline{09}, \text{null} \rangle$ causes an overflow in the middle leaf of Figure 17.7(i). This node is split into two nodes, and the pair consisting of the middle record $\overline{10}$, together with the pointer to the node allocated by the split are added to the parent node, which is the current root. As the root is full, it is also split into two nodes, and the middle record $\overline{10}$ together with the pointers to the two split nodes are stored in the new root. Figure 17.7(j) shows the final state of the tree after all the twenty insertions.

17.6. Performance of insertion in a B-tree

This section gives an informal and somewhat imprecise argument to estimate the cost of insertion. We will assume that we start with the empty tree, and insert one key at a time into it. Normally, in life of the B-tree some deletions would also occur, but we will ignore them here.

Let's assume that the height of the B-tree is h . Page accesses during execution of BTreeInsert can be logically divided into the following three phases:

- **The search phase.** This consists of starting from the root, and following child pointers along a path, P , until a leaf is reached. (See Figure 17.5.) This phase requires h page accesses.
- **The splitting phase.** If the leaf is not full, it can absorb the insertion and this phase terminates. Otherwise, the leaf is split and a $\langle \text{record}, \text{pointer} \rangle$ pair moves

upwards along the path, P , possibly splitting some nodes. We denote the number of nodes being split by s . A split involves 2 page accesses to write the 2 nodes after a split has been composed in the buffers. If the node being split is a non-root node, an additional page access is required to read of the parent node, whose pointer is found on a stack. On the other hand if the node being split is the root, there is no parent node to be read, but a new root is composed in a buffer and written to the disk. As will be clear shortly, the writing of a the new root node is not counted toward the cost of the split. In summary, if all of the s splits are for the non-root nodes, the cost of splits is $3s$, and if the root is involved then it is implied that all the h nodes in the path P have been split and the cost of splits is $3h - 1$.

- **The final write phase.** During this phase a single node is written back to the disk. If there are no splits, this node would be a leaf node. If a new root is created, it would be for writing the new root. Otherwise, it is the last parent that is updated. The cost of this phase is 1 page access.

Now we summarize the cost of an insertion. If there are s splits and the root is not split, the total cost of the insertion is $h + 3s + 1$. In the worst case the root is split, it is implied that all h nodes in the path P have been split and the cost is $h + (3h - 1) + 1 = 4h$. In the best case $s = 0$ and cost of insertion is $h+1$.

We observe that the variation in the cost of insertion in the best case ($h+1$) and the worst ($4h$) is rather large. Fortunately, the splits are rare if the fanout $p+1$ is fairly large, bringing the average cost of insertion much closer to the best case. We give a somewhat imprecise argument to estimate the average number of splits. We assume that we are starting with an empty tree, and keep on inserting records, and while doing this no deletions occur. Let's consider the following parameters:

- S = total number of splits
- S_{av} = average number of splits per insertion

It is easy to prove the equation $S = N - h$. Initially when $N = 1$ and $h = 1$, the number of splits $S = 0$. Therefore, $S = N - h$ is satisfied. Subsequently, with every split both sides of the equation increase by 1. With every new split, the increase in S by 1 is obvious. The node being split is either a non-root node or a root node. In the former case N increases by 1, and h remains the same, and in the latter N increases by 2 and h increases by 1. In both cases $N - h$, the right hand side of the equation increases by 1. Therefore,

$$S_{av} = \frac{S}{n} = \frac{N - h}{n}$$

Now, $n \geq 1 + (N - 1)d$; this is also easy to see: the root has at least 1 key, and each of the remaining $N - 1$ nodes has at least d keys.

Thus, the average number of splits is bounded by $2/p$, and the average cost of insertion is shown below. Assuming a reasonably large value of p , the performance

$$S_{av} \leq \frac{N-h}{n} \leq \frac{N-h}{1+(N-1)d} \leq \frac{N-1}{1+(N-1)d} \leq \frac{1}{\frac{1}{N-1} + d} \leq \frac{1}{d} \leq \frac{2}{p}$$

is close to the best case.

| |
|---|
| Average cost of insertion = $h + 6/p + 1$ |
|---|

17.7. Deletion in a B-tree

In this section we discuss deletion of a record from a B-tree. Deletion of a record from a B-tree is accomplished by removal of a record pointer pair from a node in the tree. Sometimes the removal of a record will cascade restructuring of the tree in the direction toward the root of the tree.

Recall that every node in a B-tree, except possibly the root, contains $\lfloor p/2 \rfloor \cdot p$ records. We say that a node *can spare* a record if it has at least $\lfloor p/2 \rfloor + 1$ records if it is a non-root node, and it has at least 2 records if it is the root node. Obviously, a node cannot spare a record if it is a non-root node and it contains exactly $\lfloor p/2 \rfloor$ records, or if it is the root node and it contains exactly 1 record. If a record pointer pair is removed from a node that cannot spare a record, a *shortage* of a record is said to be caused. A shortage leads to a temporary violation in the integrity of a B-tree that has to be fixed before deletion of a record can be considered as completed.

An understanding of deletion in a B-tree is reached by examining different scenarios surrounding a shortage. The shortage is fixed by restructuring of the tree. In order to save page accesses, the scenarios for the shortage are classified with an emphasis to localize the restructuring as much as possible.

Suppose \bar{x} is a record in a B-tree. Then a record \bar{y} in the tree is called *successor* of \bar{x} if the key value y is the smallest of all keys present in the tree that are larger than x . In the B-tree of Figure 17.8, the record $\bar{10}$ is in the root node, its successor, $\bar{11}$, is in the 4th leaf node. To reach the successor of a record, one starts with the child pointer to the right of the record, and then follows left most child pointers until a leaf is reached.

It is interesting to note that for every record \bar{x} in a B-tree, either \bar{x} or its successor is in a leaf node. This property allows us to pretend that a record to be deleted from the tree is always to be found in a leaf node. If the record to be deleted is not in a leaf node, it is swapped with its successor. After the swap, the record to be deleted is in a leaf node. This would violate the integrity of the B-tree in that the record \bar{x} is not in correct place in the tree. This problem is fixed by removing the record \bar{x} together with the pointer to its right from the leaf node. However, the leaf node may become short of a record. The scenarios surrounding this shortage and its remedy will be discussed shortly.

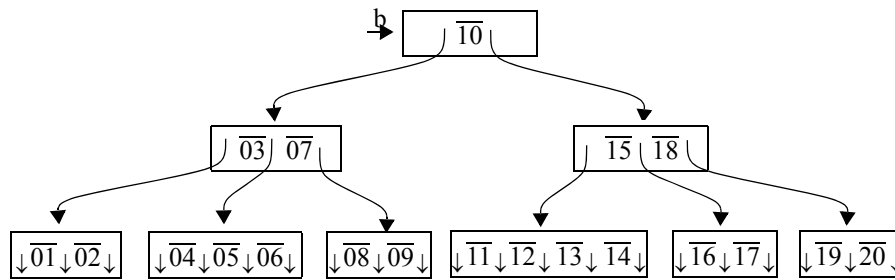


Figure 17.8. A B-tree.

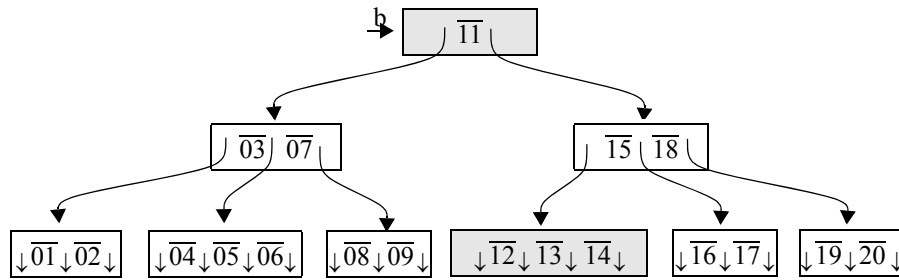
Observations: (a) The record $\overline{10}$ is not in a leaf but its successor, $\overline{11}$, is in a leaf. (b) The 2nd and 4th leaves have a record to spare. (c) The 1st, 3rd, and 5th leaves cannot spare a record, but have a sibling that can. (d) The 6th leaf and its sibling cannot spare a record. The same is true of each node at the first two levels.

We will use the B-tree of Figure 17.8 as a running example. It illustrates the concept of a successor and enumerates scenarios surrounding the potential shortage of record when removal of a record pointer pair from a node is to be attempted. Some of these scenarios requires us to examine siblings of a node. *Siblings* of a node are the children of its parent. The immediate siblings are of particular interest to us. *Immediate siblings* are siblings to the left or right of a node. Note that the left most child of a parent has only right sibling and the right most child of a parent has only left sibling, and all other children have left and right siblings.

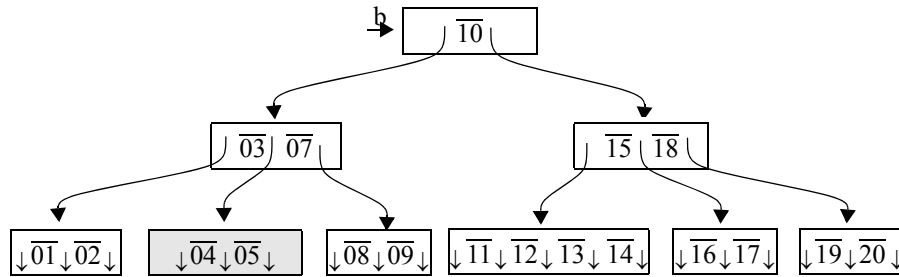
17.7.1. The algorithm for deletion in a B-tree.

The algorithm for deletion of a record \bar{x} from a B-tree proceeds in two phases. The algorithm is illustrated in Figure 17.9. The figure illustrates a variety of scenarios for deletions. The reader is cautioned that every deletion is done starting from the original state of the B-tree in Figure 17.8 in isolation of other deletions. In the first phase we start from the root and search for \bar{x} in the tree. If x is found in a leaf, the search is ended. On the other hand, if \bar{x} is found in a non-leaf node, the search is continued for its successor reaching a leaf node. The record \bar{x} is then swapped with its successor so that \bar{x} is in a leaf. This is illustrated in Figure 17.9(a).

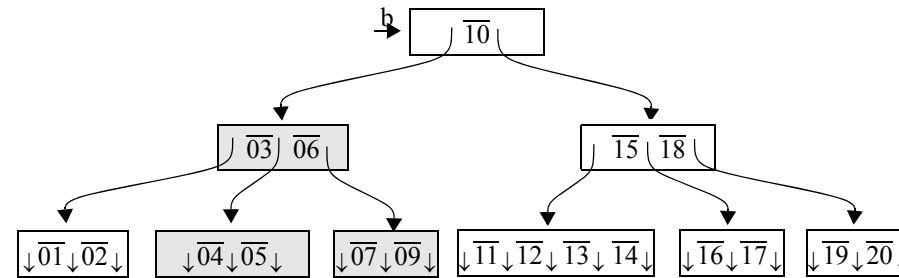
In Phase 2 of deletion, the record \bar{x} and the null pointer to its right are removed from the leaf. If this removal does not cause a shortage of a record in the leaf, the deletion is considered as completed. This scenario is illustrated in Figure 17.9(b). On the other hand, if the removal of a record pointer pair causes a shortage of a record in the node, the shortage is fixed by a local restructuring of the tree. Sometimes, this restructuring has to be cascaded all the way to the root.



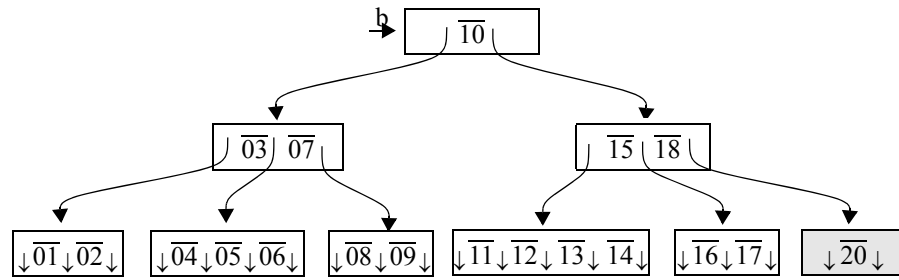
(a) Deletion of $\overline{10}$ that resides in B-tree of Figure 17.8 in a leaf node. The record $\overline{10}$ is swapped with $\overline{11}$. Now $\overline{10}$ is in a leaf. In this case, the leaf has a record to spare; therefore, $\overline{10}$ and the pointer to its right are removed. In general the removal of a record and the pointer may be as in (b), (c), or (d) below.



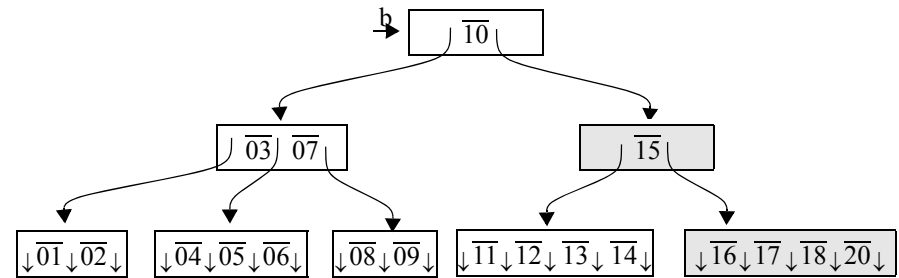
(b) Deletion of $\overline{06}$ that resides in B-tree of Figure 17.8 in a node that can spare a record. The record $\overline{06}$ and the pointer to its right are simply removed from the tree.



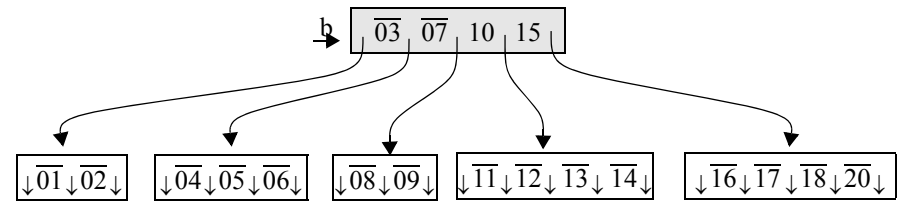
(c) Deletion of $\overline{08}$ that resides in B-tree of Figure 17.8 in a node whose sibling can spare a record. The record $\overline{08}$ and the pointer to its right are removed from the 3rd leaf, creating shortage of a record. The shortage is fixed by rotation of records: $\overline{07}$ is moved from the parent to the 3rd leaf, and the vacancy in the parent is filled by drawing $\overline{06}$ from the 2nd leaf. Rotation does not a movement of child pointers in the affected nodes.



(d) Deletion of $\overline{19}$ that resides in B-tree of Figure 17.8 in a node that cannot spare a record and does not have a sibling that can spare a record. The record $\overline{19}$ and the pointer to its right are removed. This creates a shortage of a record in the leaf. The (only) sibling to its left cannot spare a record. Continued ...



The two shaded leaves and the record $\overline{18}$ from the parent are collapsed to form one node. This creates a shortage of a record in the parent node. Continued ...



The contents of three nodes at the first two level are collapsed to form a node that becomes the new root of the tree. This finishes the deletion of $\overline{19}$.

Figure 17.9. Various scenarios for deletion in a B-tree

For each deletion we start with the B-tree of Figure 17.8.
Here parts (a) to (d) correspond to the scenarios (a) to (d) listed in Figure 17.8.

The scenarios surrounding the removal of a record are listed below. In all these scenarios, we have a node L from which the record \bar{x} and the pointer to its right have to be removed. The node L is not necessarily a leaf.

- **Case 1: The node L has a record to spare.** This scenario has been discussed before, but listed here for the sake of completeness. Note that the record pointer pair to be removed from the node L that is not required to be a leaf.

In this case the record \bar{x} and the null pointer to its right are simply removed from the tree and the deletion is considered as completed. This scenario is illustrated in Figure 17.9(b).

- **Case 2: The node L cannot spare a record, but one of its siblings can.** This scenario is illustrated in Figure 17.9(c). In this case the record \bar{x} and the pointer to its right are removed from L . To fix the shortage of one record in L , a rotation is used. A rotation moves a record from the parent to node L , and a record from the sibling of L to the parent. A rotation preserves properties of a B-tree and it only involves movements of records. No pointers are moved. A rotation also completes the deletion in a B-tree without causing a structural restructuring of the tree.

- **Case 3: Neither node L nor any of its siblings can spare a record.** This scenario is illustrated in Figure 17.9(d). In this case the record x and the pointer to its right are removed from L . In this case the shortage of one record in L cannot be filled by a sibling. In this case the node L and one of its siblings are collapsed to form a single node. But this necessitates the removal of one record pointer pair from their parent. So a record pointer pair is also pulled from the parent into the combined node. This fixes the problem at the level of the node L . If the parent becomes short of a record pointer pair, the parent node becomes the candidate for the scenario of Case 1 or Case 2 and the respective measures are taken. Clearly removal of a record pointer can cascade all the way up to the root node. If the root has at least two records it will spare a record pointer pair. On the other hand, if the root has only one record the nodes being collapsed will absorb the root and will become the new root of the tree. In this case the height of the tree will reduce by 1.

17.8. B^+ -trees

The height of a B-tree is the single most important measure of its performance of the tree, a B-tree with smaller height is desirable. In a $p+1$ -ary B-tree, we are allowed to pack up to p records in a node, which entitles us to have a fanout of up to $p+1$. The greater the fanout, the smaller the height of the tree. It should be emphasized at this point that the navigational structure of a B-tree depends only upon the key of a record and is independent of the remaining attributes. Storing a record rather than a key in a node of a B-tree prevents a B-tree from having an optimal fanout. Therefore let's consider the possibility of storing keys in the tree. Then where do we store the records? There is a good solution to this: push the records to

the last level of the tree. It turns out that all the pages containing records can be linked together forming a sequence in which all records in the file appear in the logical order of the key. The resulting tree will be termed a B^+ -tree. A B^+ -tree provides reasonable performance for direct access, and excellent performance for sequential and range accesses.

A B^+ -tree consists of an *index set* and a *sequence set*. A node in an index set is called an *index node* and a node in the sequence set is called a *sequence node*. As a default, let's assume that index nodes and sequence nodes have the same physical page size (number of bytes).

Let's establish the parameter p with an *index node*, and q with a *sequence node*. An index node contains up to p keys and $p+1$ pointers, and a sequence node contains up to q records and 1 sequence pointer. More specifically, a non-root index node contains $\lfloor p/2 \rfloor$ keys, and $\lfloor p/2 \rfloor + 1$ pointers, and a sequence node contains $\lfloor q/2 \rfloor$ records and 1 sequence pointer. The height of the tree, denoted h , includes the sequence set. Therefore, if the sequence set forms the fourth level in the tree, the height of the tree is 4.

Whereas $<$ is used for child-ward navigation in a B-tree, \leq is used in a B^+ -tree until a sequence node is reached.

The cost of retrieval of every record in a B^+ -tree is same as the height of the tree. This is different from B-trees where a record may be found in a non-leaf node in fewer than h accesses. The height, h , of a B^+ -tree is given as follows:

$$h \leq 2 + \log_{\lfloor p/2 \rfloor + 1} \frac{1}{2}(n / \lfloor q/2 \rfloor)$$

Example 17.4. Suppose a file contains 1,000,000 records, a record consists of 100 bytes, and a key value consists of 4 bytes. Let us consider 1024 byte pages. For the sake of simplicity, assume that all pointers consist of 4 bytes. We consider B-tree and B^+ -tree organizations for the file.

Let's first consider a B-tree for the file. With 100 byte records and 4 byte pointers, $p = 9$ can be chosen for 1024 byte pages. It is easily shown that the height of this tree is between 7 and 9. Thus, the performance of retrievals for this tree is between 7 and 9, that seems hardly acceptable.

Next, consider a B^+ -tree. First, we need to determine p and q . A sequence node of 1024 bytes should be able to hold up to q records of 100 bytes and a link pointer of 4 bytes. Therefore, q is easily seen to be 10. An index node of 1024 bytes should be able to hold p keys of 4 bytes and $p+1$ pointers of 4 bytes. Therefore, p is set to 127. Hence, a non-root node of the tree has 63..127 keys and its fanout is 64..128. The height of a B^+ -tree is easy to determine by estimating the number of nodes in

the tree, bottom up, level by level. In order to determine the range for the height, we try thinnest and thickest trees. These calculations are shown in the table in Figure 17.10.

| Level L | Number of nodes at level L | |
|------------|---------------------------------------|--------------------------------------|
| | The thinnest tree | The thickest tree |
| h | $\lfloor 1000000/5 \rfloor = 200,000$ | $\lceil 1000000/10 \rceil = 100,000$ |
| h-1 | $\lfloor 200000/64 \rfloor = 3,125$ | $\lceil 100000/128 \rceil = 782$ |
| h-2 | $\lfloor 3125/64 \rfloor = 48$ | $\lceil 782/128 \rceil = 7$ |
| h-3 | 1 | 1 |
| Conclusion | $h \leq 4$ | $h \geq 4$ |
| | The height of the tree is 4 | |

Figure 17.10. Height of B⁺-tree of Example 17.4.

It turns out that the height of this B⁺-tree with 1,000,000 records is always 4. Therefore, the performance for retrievals is 4 page accesses. If we are willing to replicate the root of the tree in a buffer, we can improve the performance by 1 disk access. An interesting observation is that the number of nodes in the first two levels of the tree range from 8 in the best case to 49 in the worst case. If we are willing to replicate these nodes in the main memory, a performance of 2 page accesses can be guaranteed. Note that this requires 8 to 49 buffers.

Clearly, the B⁺-tree provides a much better performance than the B-tree. At some expense of main memory, a performance of 2 page accesses could be guaranteed for retrievals.

17.9. Insertion of a record

A B⁺-tree consists of a sequence set and an index set. The sequence set is a linked list of nodes (pages), each containing $\lfloor q/2 \rfloor..q$ records. The index set is just like a B-tree except that the index nodes contain keys and child pointers instead of records and child pointers. A non-root index node contains $\lfloor p/2 \rfloor..p$ keys and $\lfloor (p+1)/2 \rfloor..(p+1)$ pointers. The root node is an index node and it contains $1..p$ keys and $2..(p+1)$ pointers.

The insertion of a record \bar{x} in a B⁺-tree is accomplished in three phases In Phase 1 one identifies the appropriate leaf node in the sequence set. In Phase 2 the record is added to the sequence set. If the leaf node has space for a record, the record is added to it and the insertion is completed. Else, splitting is necessary. The record is collated with the leaf and the resulting sequence node is split into two

nodes called the left node and the right node. In Phase 3, the pair consisting of the key of the highest record in the left node and the pointer to the right node are added to the parent of the original leaf in a way similar to that in B-trees. During this phase the splits can cascade upward and in the worst case a new root will be created. The three phases are described below in greater detail.

Phase 1. Identify a leaf node. In this phase one starts with the root of the B^+ -tree and follows child pointers reaching a leaf node. This navigation is similar to that in B-trees. In the node being visited, the smallest key k satisfying $k \leq x$ is identified, where x is the key of \bar{x} . If such a key exists, the child pointer to the left of the key is followed. Otherwise the last child pointer in the index node is followed. This will eventually lead to a leaf node in the sequence set. At the end of this phase, the contents of the leaf node are in a buffer.

Phase 2. Add record to the sequence set. If the buffer obtained from the previous phase has space in it, the record \bar{x} is added to it. The buffer is then written to the disk in place of the original leaf, and insertion is completed.

On the other hand, if the buffer obtained at the end of Phase 1 is full (containing q records) a split will be composed using two buffers. For convenience, we refer to the two buffers as the *left buffer* and the *right buffer*. To compose the split, first then buffer obtained from Phase 1 and the record \bar{x} are collated. The result is an imaginary overflow buffer. Now a split is composed in the two buffers by splitting this node so that each buffer gets one half of the records. If q is odd, we let the left buffer have one record more than the right buffer.

It is our intention to write the left buffer in the place of the original leaf node and link it to a new node containing the records from the right buffer. The new node should point to where the original leaf node was pointing. To accomplish this, a new node is allocated and the link pointer of the original leaf from Phase 1 is transferred to right buffer. The address of this newly allocated node becomes the link pointer in the left buffer. The composition of the split is now complete. The two buffers are then written to the disk.

Next the middle record of the overflowing node is designated. For this algorithm, the middle record is defined to be the highest record in the left buffer after the split.

Phase 3. Add key pointer pair to the index set. Now a key pointer pair is formed. The key is the copy of the key of the middle record identified at the end of Phase 2. The pointer is the address of the right node allocated in Phase 2. Now this key pointer pair is added to the parent of the original leaf node from Phase 1. From here onward, the addition of key pointer pairs is identical to that in B-trees. The only difference is that in a B-tree one adds record pointer pairs, but here key pointer pairs are added.

Example 17.5. Now we give example of insertions in a B^+ -tree with $p = 4$ and $q = 3$. The root is always an index node, and it can contain 1..4 keys and 2..5 pointers. Other index nodes can contain 2..4 keys and 3..5 pointers. A sequence node can hold 1..3 records. When we have a full sequence node with 3 records and add a new record to it, the sequence node splits into two nodes, each containing 2 records.

Starting with an empty B^+ -tree, we insert the records $\overline{02}, \overline{17}, \overline{05}, \overline{09}, \overline{13}, \overline{03}, \overline{11}, \overline{01}, \overline{04}, \overline{15}, \overline{08}, \overline{19}, \overline{14}, \overline{16}, \overline{18}, \overline{12}, \overline{20}, \overline{07}, \overline{06}, \overline{10}$ in a sequence. The empty tree simply consists of a null pointer (to an index node). When $\overline{02}$ is inserted, one obtains a B^+ -tree with one node in the sequence set and one node in the index set. The sequence node contains the record $\overline{02}$ and a null link pointer. The index node contains a pointer to the sequence node followed by the key value of 02. This is a small tree and it does not conform to our definition of a search tree. For example the index node contains 1 key and not 2, but 1 child pointer. Strictly speaking we should allow such exceptions to be built into the definition of B^+ -tree. This is a bit tedious but simple. We ignore this detail for small states of the B^+ -tree. Figure 17.11 shows the results at several stages in the sequence of insertions.

17.10. Performance of insertion.

Analysis of performance of insertion in a B^+ -tree is similar to that in a B-tree. If there are s splits and the root is not split, the total cost of the insertion is $h + 3s + 1$. In the best case $s = 0$ and cost of insertion is $h + 1$. In the worst case the h nodes are split, including the root, and the cost is $4h$. On average, we can assume that $s = 2/q$. Therefore, the average cost of insertion is $h + 6/q + 1$, which is close to the best case performance (and much better than the worst case).

17.11. B^+ -tree as a secondary index

Every file structure can be used either as a primary structure or a secondary structure. This is also true of B^+ -trees. When a B^+ -tree is used as an indirect index, a sequence node contains up to q $\langle \text{key, primary page pointer} \rangle$ pairs, and 1 sequence pointer. The index nodes in a secondary B^+ -tree are similar to the index nodes in a primary B^+ -tree. More specifically, a non-root index node contains $\lfloor p/2 \rfloor..p$ keys and $\lfloor p/2 \rfloor + 1..p + 1$ pointers, and a sequence node contains $q/2..q$ keys, the same number of primary page pointers, and 1 sequence pointer.

As in primary B^+ -trees, we will assume that index and sequence nodes in a secondary B^+ -tree use the same size pages. Sometimes we can assume that a page pointer has the same physical size as an internal pointer. Under this assumption $p = q$. Otherwise p and q may be different. The height h of the secondary tree is same as the level of the sequence set. Therefore, if the sequence set is at the fourth level of the tree, the height of the tree is 4. This means that at the 4th level we only have

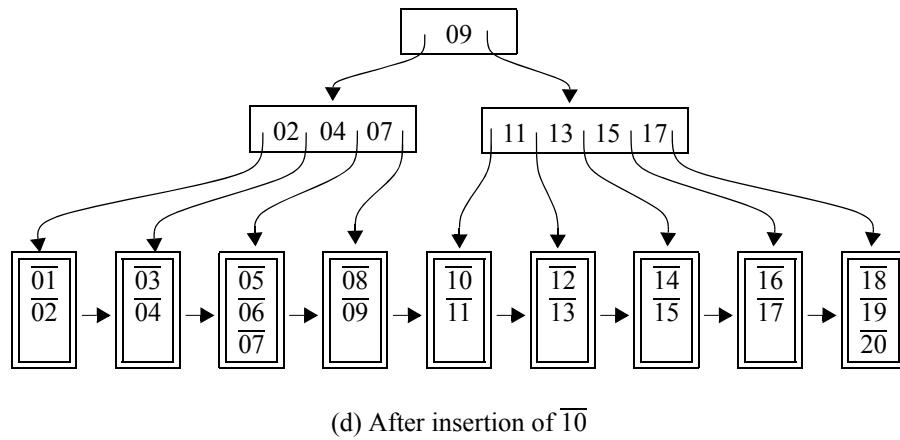
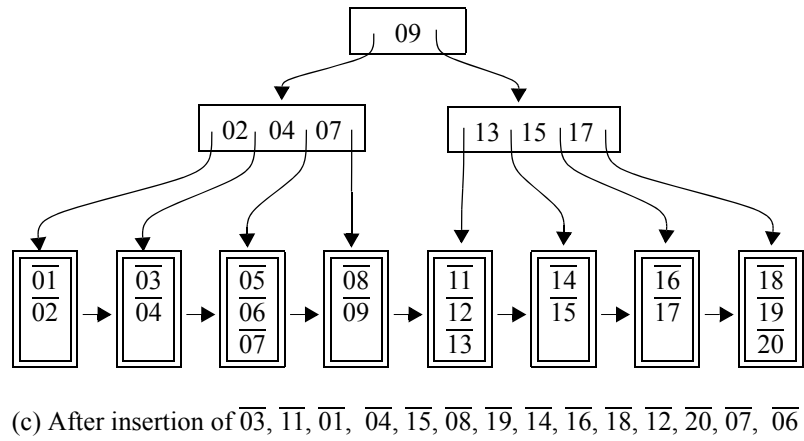
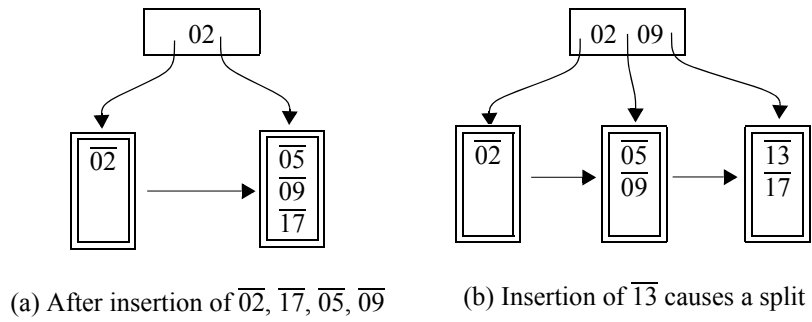
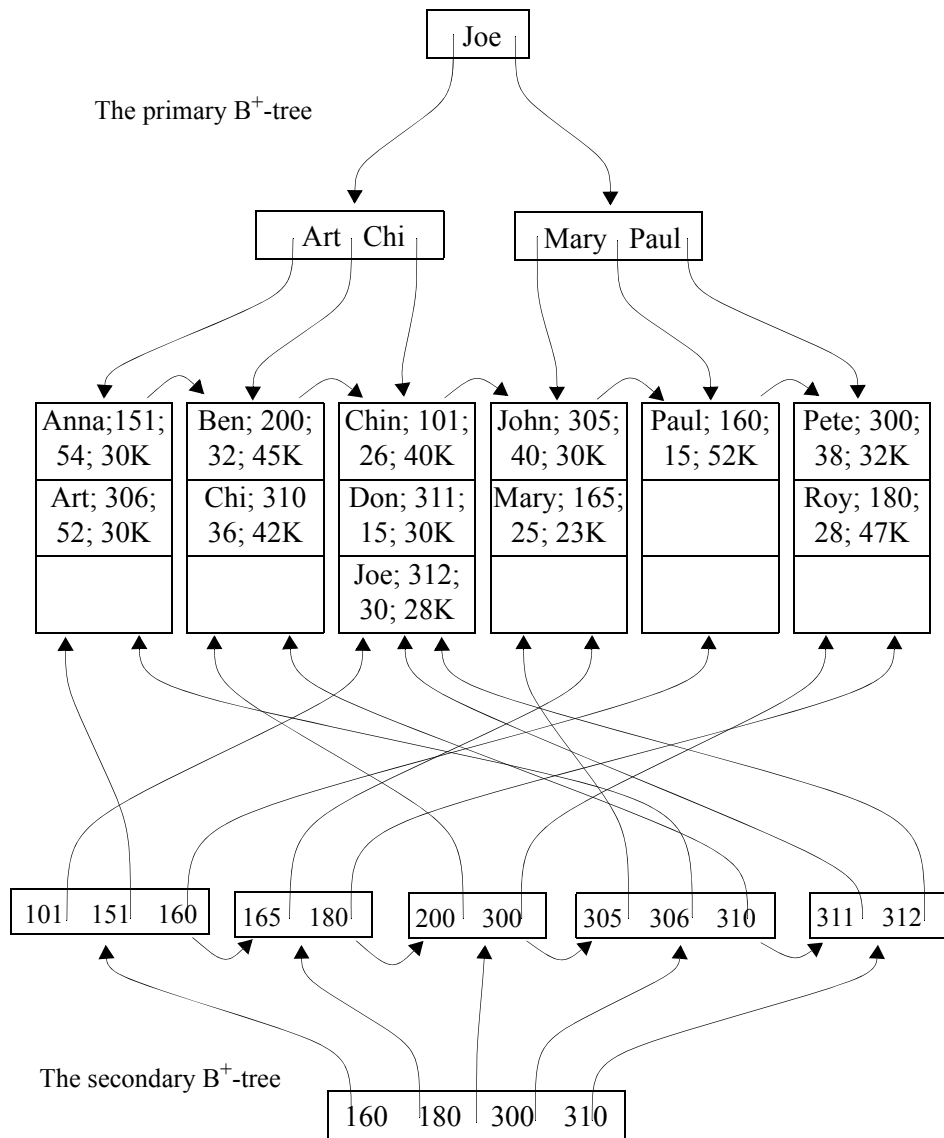


Figure 17.11. Insertions in a B⁺-tree
pointers to the records; records themselves are elsewhere in a primary file that is separate from the secondary B⁺-tree at hand.



The emp file on Page 56 is the sequence set of B⁺-tree with Name as the primary key ($p = 4, q = 3$).
 The B⁺-tree is a secondary index by ID ($p = q = 4$).

Figure 17.12. Primary and secondary B⁺-trees

Example 17.6. Figure 17.12 shows an example of a primary B⁺-tree and a secondary B⁺-tree. The emp file on Page 56 forms the sequence set of the primary B⁺-

tree. In this tree $p = 4$ and $q = 3$. Therefore, a sequence node can hold 1..3 records and 1 sequence pointer, and an index node (except the root) can hold 2..4 keys and 3..5 pointers. The B^+ -tree is a primary index.

We can also install a secondary index by ID organized as a B^+ -tree. In this tree $p = q = 4$. Therefore a sequence node in the B^+ -tree can hold 2..4 ⟨key, primary page pointer⟩ pairs and a sequence pointer. An index node can hold up to 4 keys and 5 pointers.

Appendix A

On nature of implication in reasoning

When we carry out reasoning in a natural language such as English, certain rules of deduction are implicitly assumed. These rules of deduction are formalized in formal systems of logic. The rules related to connectives \vee , \wedge , \neg , \Rightarrow , and \Leftrightarrow are formalized in propositional calculus, and the rules for quantifiers \exists , \forall are formalized in predicate calculus. A natural language can be viewed as a consistent extension of these formal systems of logic; therefore, a clear understanding of the formal rules of deduction is important.

Perhaps the only rule that seems confusing to a beginner is the definition of implication (\Rightarrow). The following table shows the definitions of implication (\Rightarrow) and equivalence (\Leftrightarrow) in propositional calculus. The correct value of * in the table is True, the main purpose of this handout is to explain why this is so.

| P | Q | $P \Rightarrow Q$ | $P \Leftrightarrow Q$ |
|-------|-------|-------------------|-----------------------|
| True | True | True | True |
| True | False | False | False |
| False | True | * | False |
| False | False | True | True |

The role of implication in reasoning is to deduce facts: if we know that P and $P \Rightarrow Q$ are true, we conclude Q is true. A priori, if P is known to be true, the case in third row of our table does not even arise. Thus we note that the choice of True for * cannot lead to any inconsistency.

If we replace * with False, we would fail to differentiate between implication and equivalence. This would have undesirable consequences. For example, consider the true statement "*ABCD is a square \Rightarrow ABCD is a rectangle*". If we do not differentiate between implication and equivalence, either we will fail to accept the truth of the above statement, or will have an uninteresting geometry where every rectangle would be a square. Clearly, neither of the two alternatives is attractive.

Now we argue why * must be True through an illustration. In set theory, we like to have $A \cap B \subseteq A$ as a theorem. What happens if $A \cap B$ turns out to be \emptyset , the empty set? Clearly, $\emptyset \subseteq A$ should be true for every set A.

Let us examine the consequences of having $\emptyset \subseteq A$. As an example, suppose $A = \{1, 2\}$. To prove $\emptyset \subseteq A$, we need to show that for an arbitrary element x the statement " $x \in \emptyset \Rightarrow x \in A$ " is true. Let us consider the cases $x = 1$ and $x = 3$ in

more detail. The following table shows these cases.

| x | $x \in \emptyset$ | $x \in A$ | $x \in \emptyset \rightarrow x \in A$ |
|---|-------------------|-----------|---------------------------------------|
| 1 | False | True | True |
| 3 | False | False | True |

Note that if we want $\emptyset \subseteq A$, every entry under $x \in \emptyset \Rightarrow x \in A$ must be True. But then the first row of this table clearly says that * in the previous table, we must be a True.

Appendix B

Disk Architecture

In this chapter we discuss the physical architecture of disks in order to gain an understanding of performance issues for access of large amounts of data.

Main memory is a premium resource and relatively small in size. Therefore, large amounts of data is stored on the disk. In order to process data, CPU does not have direct access to the disk – the data has to travel between the CPU and the disk via main memory. A *read* operation brings data from the disk to the main memory and a *write* operation does the reverse. A disk *access* is either a read or write operation. As a disk is partly a mechanical device and disk access requires a long delay. On the other hand CPU can process data at a fast speed. Because of the gap between the speeds of the disk and CPU it becomes necessary to use large chunks, called blocks, as smallest units for disk accesses. The operating system fixes a fixed block size and has the same size buffers in main memory to facilitate disk accesses. The size of a block in bytes consists of a power of 2, typically $\frac{1}{2}$, 1, 2, 4 or a larger number of kilobytes. All I/O is in terms of reading and writing blocks.

Whereas the speed of CPU is increasing steadily the overhead associated with a disk access has improved only marginally. Therefore, the differential between the speeds is increasing. Under such conditions one would expect that the block size should increase at a pace that is linear relative to the growing differential in speeds. But an operating system has to deal with very small to very large files. A large number of the files are very small and choosing a large block size is a potential for large amounts of unused space due to fragmentation. On the other hand, a database has to deal with small number of large artifacts such as relations and indexes. In addition, stream oriented processing is a staple in databases. These considerations suggest leaning toward much larger units for disk accesses. We have termed this unit as *page* to differentiate it from a block, a unit of disk access for an operating system. (The reader should note that the terms "block" and "page" are used interchangeably in the literature.) A page in database systems consists of a multiple blocks of the operating system. Again this multiple is usually a power of 2. Thus a page could consist of 1, 2, 4, 8 blocks. A page size of 16 kilobytes is not uncommon.

In order to achieve a better throughput, often database systems like to do their own management of disk space and disk accesses rather than depending upon the operating system. But as we will see in this chapter all page accesses do not cost the same. We will see that, e.g., accessing 10 pages on the disk at once is much faster than accessing them 1 page at a time. But this requires a better understanding

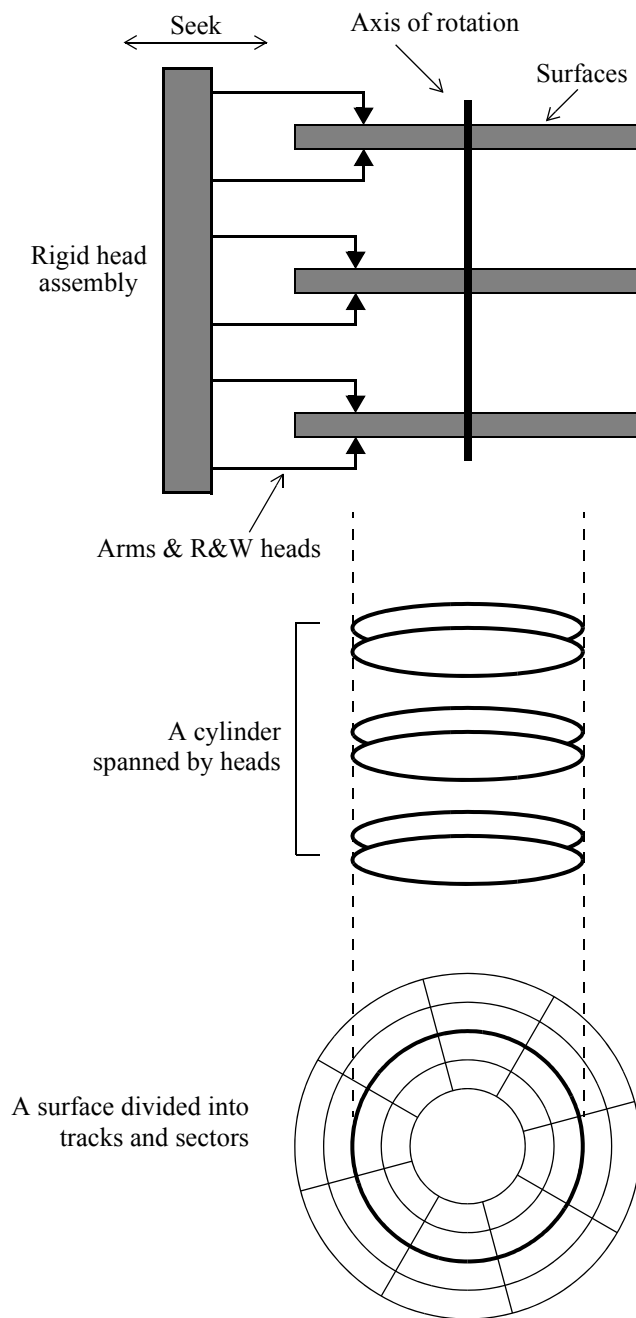


Figure B.1. Disk

of the disk and its management requires a good understanding of the nature of the address space on the disk.

B.1. Physical composition of a disk

The composition of a disk is shown in Figure B.1. A disk contains one or more concentric circular *surfaces*. These surfaces constantly rotate around a common axis. Each surface has a read / write head dedicated to it that is mounted on an arm. A head at different positions spans circles on a surface that are called *tracks*. Tracks are partitioned into what are called *sectors*. The marks identifying boundaries of sectors are created when the disk is formatted. The size of a sector is chosen so that its capacity in bytes is same as a block in an operating system. (Recall, that the size of a block is itself fixed by the operating system.)

As stated above, all the heads are mounted on arms. All the arms are connected to a common rod. Thus the heads form a rigid assembly – they cannot move independently on different surfaces. All of them move together, always remaining in a straight line relative to each other. At any given time the heads are on concentric tracks on different surfaces. Only one head is active in reading or writing, but switching from a head on one surface to a head on another is instantaneous. When put together these tracks for a fixed position of head assembly are said form a *cylinder*. The lateral movement of the head assembly, called a *seek*, is a mechanical one, hence very slow. Therefore, the concept of a cylinder is important. All the sectors within a given cylinder are accessible without requiring intervening seeks. Thus sectors within a cylinder are very “close” to each other.

The purpose of a seek operation is to “seek” a cylinder where the desired sector resides. The time it takes to seek is denoted as s . After a seek has been completed, one has to wait for beginning of the sector to align with a head. This part of the mechanical movement is called rotational latency. Recall that head on any surface can be activated instantaneously. Therefore, for a given sector there is a sector on every track of a cylinder that can be traverse immediately without any intervening rotational latency.

It can be assumed that while a track is being traversed, transfer of information between the disk and main memory is without any delay. We assume that a page in a database consists of one or more blocks that are stored contiguously without gaps within a cylinder. Thus after having gone through a seek and latency operations the last component, called *transfer* time, is the time it takes for the surface to rotate one page. We denote this as p , the time needed to transfer one page. Then the time required to access a page from the disk is given as $s + l + p$.

B.2. Cost of page accesses

Recall s , l , and p denote the time to seek a cylinder, rotational latency to align

with the beginning of a page, and the time it takes to transfer a page (to or from) memory, respectively. Now suppose we need to access m pages. We note that irrespective of the relative placement of pages $m \times p$ milliseconds will be spent in the overall transfer operation. The rest is the overhead or delay caused by seeks and rotational latencies. With this in mind, the time required to access these pages depends upon their relative positions on the disk. These are listed as sums of transfer times and overheads as follows.

- $m \times (s + l + p) = m \times p + [m \times (s + l)]$, if the pages are scattered on the disk
- $s + m \times (l + p) = m \times p + [s + m \times l]$, if the pages are in the same cylinder
- $s + l + m \times p = m \times p + [s + l]$, if the pages are contiguous

Example B.1. Suppose a disk with seek time (s) of 10 ms and latency (l) of 5 ms. Suppose pages of a given size require a transfer time (p) of 2 ms. Consider $m = 100$ pages. The overall transfer time for these pages is $100 \times 2 = 200$ milliseconds. Depending upon the relative placement of these pages on the disk the costs of their access are as follows.

- $200 + [100 \times (10 + 5)] = 200 + [1500] = 1700$ ms for pages scattered on the disk
- $200 + [10 + 100 \times 5] = 200 + [510] = 710$ ms for pages in the same cylinder
- $200 + [10 + 5] = 200 + [15] = 215$ ms for pages that are contiguous

B.3. Multiplicity of access patterns

The data on a disk always has some *stored physical order* whether a specific order, such as the one imposed by a B-tree, is intended or not. However, the data may be accessed in one of many possible patterns that are initiated due to different reasons ranging from the wishes of an end user to that of a query processor that may choose to invoke a specific algorithm for achieving efficiency. It is important to keep in mind that where as there is only one stored pattern, there may be multiple types of access patterns. We should expect that the cost of access will vary from one type of access pattern to another. Let's enumerate a range of possibilities. In order to understand this issue, let's keep in mind that the cost of access consists of seeks, rotational latencies, and transfer times. As observed before, of these, the transfer time depends only upon the amount of information being transferred and independent of the access pattern. Therefore, our main concerns are seeks and rotational latencies of which the former is generally more troublesome.

- One gets the best performance if the access pattern is same as the stored pattern. In many streaming operations this is actually the case. An example of this would be computation of total payroll, the sum of salaries of all employees. In the best performance would be achieved if the pages form a cascade. Considering that we do not mind the transfer time, the overhead is only one seek and one rotational delay that is negligible if large amounts of data is accessed. However, storing artifacts as

a cascade under updates can cause greater fragmentation and increased overhead for maintenance of the artifact.

- The second best alternative is that the access patterns expects the records to be encountered in certain order and if that order is guaranteed in the storage as one goes from one page to another, but the pages are not necessarily contiguous. An example of this arises when we want to scan an Emp relation in order of names and the relation is actually sorted by Name-values, say as a sequence set of a B+ tree (see Page 65). Note that the pages (nodes) in the sequence set may be scattered on the disk and hopping from one node to another may require a seek and a rotational latency. If the sequence set resides on the same cylinder seeks can be saved. Note that the logical order considered in this example can arise because of the wishes of the user or an internal algorithm to process a natural join. If m pages are accessed, in addition to the transfer time of $m \times p$ the overhead is $m \times (s + l)$ if the pages are scattered on the disk and $m \times l$ if pages are in the same cylinder or smoothly roll over in neighboring cylinders.
- The worst alternative is when the access pattern is random relative to the storage pattern. An example of this is when a secondary index is used to scan a relation in logical order of the secondary key where each record requires one or more disk accesses. For example, recall the secondary index by ID-values on Emp relation in Chapter 5 (see Page 63). In this example every record requires a seek and a rotational latency. This is much worse than requiring the overhead for accessing each page if the size of records is much smaller relative to pages.

Although this is the worst case among the three enumerated here, it is not without merits. Recall that a file can have only one physical order that is sometimes designated as primary. Often other access structures are desired. As multiple secondary structures can coexist, they allow several independent access patterns to be realized simultaneously. Sometimes an access pattern may be colinear with one of the secondary structures. In order to put the above observations in perspective, here is an example.

Example B.2. Let's assume a disk with same characteristics as in Example B.1. Suppose a file contains 30,000 records and that 48 records are stored per page. Clearly, the number of pages needed by the file is $30000/48 = 625$. Now suppose we want to read the entire file. Several access patterns are enumerated below.

(a) Read the whole file as a cascade. There are 625 pages in the file. We need $s + l + 625p = 10 + 5 + 625 \times 2 = 1,265$ milliseconds ≈ 1.3 seconds.

(b) Read the whole file by pages at random. We will need $625 \times (s + l + p) = 625 \times (10 + 5 + 2) = 10,625$ milliseconds ≈ 10.6 seconds. Note that if the file resides in a cylinder the cost is $s + 625(l + p) = 10 + 625 \times (5 + 2) = 4,385$ milliseconds ≈ 4.4 seconds.

(c) Read the whole file by records at random. There are 30,000 records, so we need 30,000 random reads. If the records are scattered on the disk, this requires $30000 \times (10 + 5 + 2) = 510,000$ milliseconds = 510 seconds. If the records are in the same cylinder, 210,000 milliseconds = 210 seconds.

We see that to access the same logical data the times required are 1.3, 4.4, 10.6, 210, and 510 seconds. The slowest access takes approximately 400 times longer than the fastest. Obviously, exploiting the physical contiguity can help considerably. But even with good physical contiguity due to multiplicity of access patterns the costs vary a lot.

B.4. The structure of address space on a disk

In most of this book we have assumed random access of pages as if they were scattered all over the disk. This helped us to have a simple model of performance that helps us in understanding performance issues and importance of buffer management in databases. This is not meant to undermine the importance of physical locality on disk. Any file structures and utilities developed for wide use should exploit the performance advantages offered by physical locality.

Considering need for streaming in databases, physical placement of pages to take advantage of locality is an important issue. It should be obvious that in order to exploit the locality as one goes from one sector to another, the address space of sectors can be conceptualized in terms of triples of addresses (Cylinder#, Track#, Sector#) in lexicographical order.

Due to increasingly finer resolution of information stored on the disks the number of cylinders as well as capacities of cylinders is increasing steadily. As databases have small number of large artifacts we may consider storing the artifacts in disjoint sets of cylinders. In other words, a cylinder may not be shared by two different artifacts. In addition, the most accessed artifacts may be stored in the middle cylinders. This would reduce seek time when jumping from one artifact to another.

Use of multiple disks is also advisable. For example, instreaming and outstreaming are often quite intertwined in terms of the sequence in which pages are read and written. If all these pages are on the same disk the seeks could take significant amount of time. Instreaming from one disk and outstreaming to another may save considerable amounts of time. In some time-consuming operations one may use a disk for each stream. An example is multiway merging to help sort large volumes of data.

B.5. RAIDS

Let's imagine a hypothetical situation that the heads on different surfaces of the disk could move independently of each other rather than having to move in one rigid assembly. Then while pages are accessed from one surface the heads on other

surfaces could take position to resume page accesses without further delay. From the point of view of performance this is precisely how one can view a RAID, redundant array of inexpensive disks. A raid consists of several disks, although heads in any disk move as a rigid assembly, but across disks they can be considered independent. RAIDs are a good and inexpensive alternative to achieve a speedup for large cascades of pages. In general RAIDs provide one to achieve a combination of increase in storage capacity, performance, and fault tolerance under disk failures.

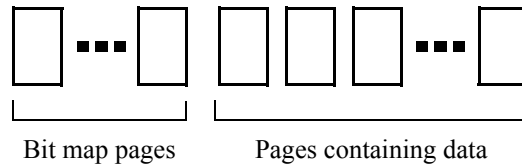
Appendix C

Management of storage, buffers, and streaming

- Storage organization
- Storage and buffer management
- Database catalog
- Page format for relations with fixed size tuples
- Stream-based access to tuples via iterators

C.1. Organization of a storage

- Storage is realized as a sequence of pages
 - (Typically) all pages are of a fixed size
 - Pages are simply arrays of bytes

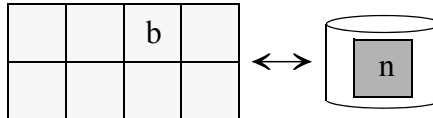


Storage organized as a heap of pages

- Storage is organized as a heap
 - A bit map is created to keep track of pages being used / free
 - The bit map for allocated pages is stored in initial pages on the disk
Allocation bit records whether a page has been allocated (1) or free (0)
 - Clients needing pages must allocate them
A free page is found, its allocation bit is set, and its address returned
 - Clients should allocate only pages that they will use
 - When a page is no longer needed, it must be deallocated
Its allocation bit is set to 0 to reflect that it is free
- Instrumentation for page allocation
 - Session-based counters are maintained to keep track of
PagesAllocated and PagesDeallocated
- Client modules must use the following functions
 - AllocatePage, DeallocatePage, ReadPage and WritePage
 - To the storage manager client pages are simply byte sequences
 - The format of pages and their contents are managed by client modules
 - End users do not directly interact with pages or even storage

C.2. Buffer management: infrastructure

- Buffers, bookkeeping information, and replacement policy
- A pool of B buffers $\text{Buff}[0..B-1]$



- A map $b \leftrightarrow n$ that indicates buffer b contains page n
- $\text{PinCount}[0..B-1]$: integer to keep track of the number of processes that are currently using the page residing in a buffer
- $\text{HasChanged}[0..B-1]$: bit that indicated a buffer has been modified. If $\text{HasChanged}(b) = 1$ it b must be written to its corresponding page before it can be used to access a different page
- A replacement policy (e.g. LRU) that finds a buffer b with $\text{PinCount}(b) = 0$ when necessary
- PageRequests : a count of the number of pages requested
- PageAccesses : a count of pages accessed from the disk
- $\text{HitRatio} = (\text{PageRequests} - \text{PageAccesses}) / \text{PageRequests}$

C.3. Buffer management: accessing pages

- Reading a page

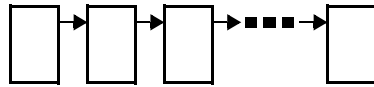
```
ReadPage(int n): Returns a buffer address b or null {
    Examine the mapping  $b \leftrightarrow n$ ;
    if n is in some buffer b {
        PinCount(b) := PinCount(b) + 1;
        return b;
        PageRequests := PageRequests + 1;
        HitRatio = (PageRequests - PageAccesses) / PageRequests }
    else use replacement policy to find a buffer b with PinCount(b) = 0 {
        if HasBeenChanged(b) = 1 {
            write buffer to its corresponding page according to  $b \leftrightarrow n$ ;
            HasBeenChanged(b) := False; }
        b := ReadPageFromDisk(n);
        PageRequests := PageRequests + 1;
        PageAccesses := PageAccesses + 1;
        HitRatio = (PageRequests - PageAccesses) / PageRequests }
        return b; }
    else return null;
}
```

- Writing a page

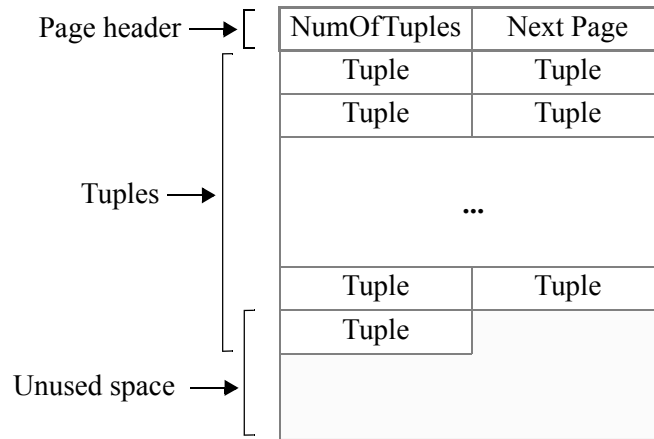
```
WritePage(int: n, byte[ ] PageContents) {
    Examine the map  $b \leftrightarrow n$ ;
    if n is not in the buffer pool {
        use replacement policy to find buffer b with PinCount(b) = 0;
        if hasChanged(b) = 1 {
            write buffer to its corresponding page according to  $b \leftrightarrow n$ ;
            HasBeenChanged(b) := 0; }
        bufferManager.replaceBuffer (b, pageContents);
        HasChanged(b) = 1;
        update the mapping  $b \leftrightarrow n$ ;
    }
}
```

C.4. Organization of pages

- Client modules know their pages
 - Storage manager knows the format and contents of only bitmap pages
 - Storage manager is oblivious to the format and contents of client pages
Client pages are simply byte sequences to the storage manager
- How a relation may be stored
 - Tuples are stored as byte sequences in a linked list of pages



- Here we assume that tuples have a uniform format and fixed length.
All attributes are stored in their native binary form
Lengths of attributes and tuples can be inferred from database catalog
- A possible page format is shown below



- A page contains a header, tuples as byte sequences, and empty space
- In general a page header records information to help decipher the binary contents of a page unambiguously. Here, it consists of the number of tuples in the page and address of the next page
- Tuples remain compacted with no gaps with fixed byte offsets

C.5. An example

- Consider the following catalog for Personnel database

```
<?XML version="1.0" encoding="utf-8" ?>
<Relational_Db dbName="Personnel" numOfRelations="2"
  <dbRel relName="Emp" numOfAttributes="3" StartPage ="5055">
    <dbAttr attrName="Name" attrType="string" attrLength="20" />
    <dbAttr attrName="DName" attrType="string" attrLength="10" />
    <dbAttr attrName="Salary" attrType="integer" attrLength="4" />
  </dbRel>
  <dbRel relName="Dept" numOfAttributes="2" StartPage ="6003">
    <dbAttr attrName="DName" attrType="string" attrLength="10" />
    <dbAttr attrName="MName" attrType="string" attrLength="20" />
  </dbRel>
</Relational_Db>
```

- Recall that the header consists of 8 bytes
- Suppose a page size of 1 kilobytes (1024 bytes)
- It has 1016 bytes available for tuples
- An Emp tuple consists of 34 bytes, so a page can hold up to 29 tuples
- An Dept tuple consists of 30 bytes, so a page can hold up to 33 tuples
- A page will have fewer tuples if space utilization is not 100%

C.6. Iterators for instreaming data

- Streaming of data is a common operation in databases
 - Iterators capture instreaming in database internals hiding cumbersome page level boundaries from clients.
 - An example is JDBC where instreaming is managed by the system
- An iterator for instreaming tuples of a relation
 - Consists of open, getNext, and close functions, and hasNext boolean
 - It can be instantiated for any relation; it will get the necessary information page from the database catalog
- Open
 - The open function reads the first page of the relation.
For example Page 5055 for Emp relation
 - It will read the number of tuples from the first 4 bytes, skip the 8 bytes of the header and position itself at the first tuple.
 - The hasNext flag is set to True
- getNext
 - It reads the byte sequence and tries to skip to the next tuple location
In case of Emp relation 34 bytes will be skipped
 - If end of page is reached, next page is read and the header is skipped.
 - If there is no next tuple hasNext is set to false
- Close closes the iterator
- Additional features
 - An iterator may have a number of buffers available to it
In that case it will read more pages at a time
 - When a relation has to be instreamed multiple times a Reset function can be implemented to bring the cursor to the first tuple

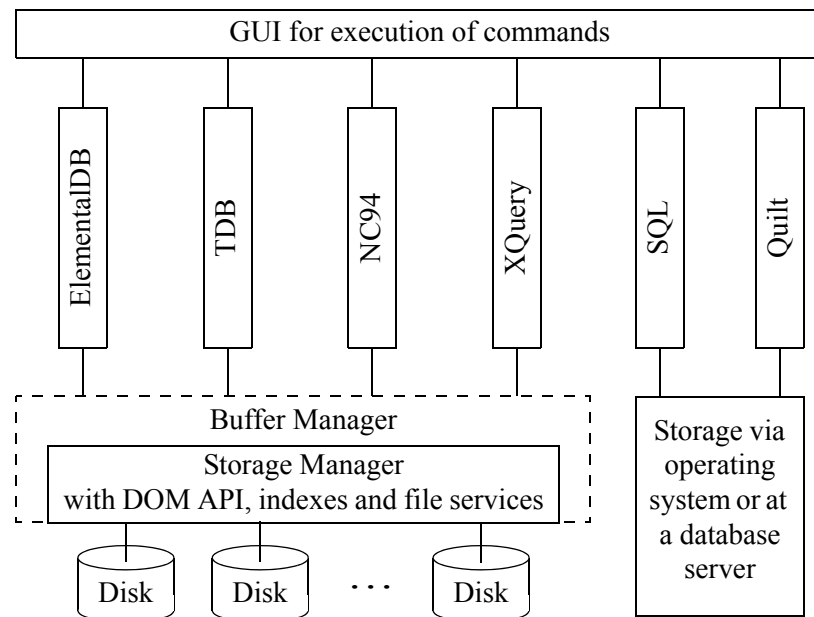
Appendix D

The Cyclone Database Implementation Workbench

- CyDIW: a multipurpose platform with a common GUI
- Supports development of new database prototypes
 - Provides infrastructural support
 - Page-based storage, buffer management, scripting, benchmarking
- Existing command based systems can be used
 - For example, SQL queries from MySQL and Oracle can be executed
 - SQL and XQuery queries can be mixed in the same batch
- Supports different types of users
 - Students can concentrate on learning
 - Instructors and teaching assistants
 - Researchers and developers
 - Also perhaps managers

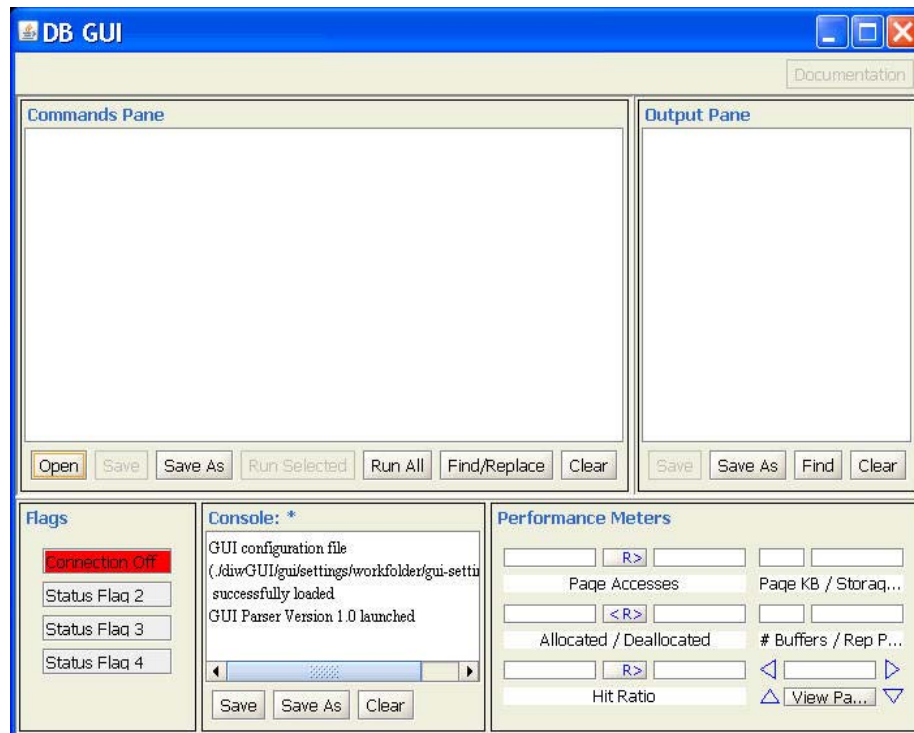
D.1. The scope of database development

- Several systems are being implemented
 - Our own
 - GUI support for others
- A user can create own storage on available disk space
 - ... megabytes to gigabytes on laptops; up to terabytes on desktops
 - Storage is a sequence of pages that are themselves byte sequences
- Buffer management is available
 - Good buffer management is critical in database performance
- The following shows organization of the workbench



D.2. GUI based environment for commands

- A picture of the GUI is shown below
- The GUI is an editor cum launch pad for batches of commands
 - Commands from different systems can be interleaved
 - They can avail services offered by CyDIW platform
- Advantages
 - Almost no learning curve
 - A batch can be created to represent a complete experiment
 - E.g., an experiment can include SQL, XQuery, and O.S. commands
 - Experiments can be quite a bit more complex and interesting
 - The experiment can be repeated in future at the click of a button



D.3. Use of XML for achieving modularity and accessibility

- Two types of XML files
- Small XML files
 - These are ordinary text-based .xml files available in operating system
 - Small .xml files are used for
 - Configurations, settings, system catalogs, and global variables
 - Complex artifacts and inter-module parameters such as syntax trees, expression trees, algebraic transformations, plans for query execution
 - These files can be processed using DOM API
 - Such files are portable across all computer systems today
- Large XML files
 - The storage manager supports .bxml and .cxml formats
 - .bxml is a binary paginated XML file stored in a tree format
 - .cxml is an XML file stored as a stream of characters in our storage
 - Many data files are stored as .bxml and processed using our own DOM API
 - We aim at terabyte range XML files

Appendix E

Armstrong Rules for functional dependencies

- Armstrong rules of deduction
 - Suppose a relation schema R is given. The following rules of deduction have been given by Armstrong.
 - Transitivity. If $X \rightarrow Y$ and $Y \rightarrow Z$ hold in R , then $X \rightarrow Z$ is also holds in R .
 - Reflexivity. If $X \supseteq Y$ then $X \rightarrow Y$ holds. (For example $ABC \rightarrow BC$.) Such dependencies are called *trivial* functional dependencies
 - Augmentation. If $X \rightarrow Y$ holds in R , then for every W , $XW \rightarrow YW$ also holds in R .
- The rules for deduction were first given by Armstrong
 - He also proved their soundness and completeness
 - In Chapter 10 we chose to presented Maier's rules
Maier's rules facilitate monotonic deductions that are much simpler than deductions using Armstrong's rules

Appendix F

Some observations on dependency theory

- The issues of redundancy seem to be rooted in the choice of the underlying database model
 - The classical relational model is presupposed
 - The model only allows atomic values
- What is the database model supported nested relations?
- Example 1. Revisit the SAIP example with $S \rightarrow A$, $SI \rightarrow P$
 - Based upon dependency theory we have the design $\{SA, SIP\}$
 - An alternative is the nested scheme $\text{supplierParts}(S, A, \text{Parts}(I, P))$
 - A state of supplierParts as a nested relation would be as follows

supplierParts

| S | A | Parts | |
|------|------|-------|------|
| | | I | P |
| ABC | Ames | Nut | 0.50 |
| | | Bolt | 0.60 |
| Acme | DSM | Nut | 1.00 |

- For an SA value, the multiple IP values consist of a nested relation
- No redundancy, a priori lossless and amenable to preservation of dependencies
- Example 2. Revisit the CAZ example with $Z \rightarrow C$, $CA \rightarrow Z$
 - A C-value has multiple Z values and a Z-value has multiple A values
 - No redundancy, a priori lossless and amenable to preservation of dependencies
- Example 3. Consider the CTHRSG example

- C, T, H, R, S, and G stand for Class, Teacher, Hour, Room, Student, and Grade
- Functional dependencies: $C \rightarrow T$, $HR \rightarrow C$, $HT \rightarrow R$, $CS \rightarrow G$, $HS \rightarrow R$
- Multivalued dependencies: $CT \twoheadrightarrow HR$ and $CT \twoheadrightarrow SG$ Hold
- We have not covered multivalued dependencies in this book
- But we note that in this case the schema $CT\{HR\}\{SG\}$ works

Appendix G

Can we use ER Model for storage and query?

- From an abstract view ER schema can be seen as a graph
 - Entity sets can be seen as nodes (ISA does not give rise to new nodes)
 - Relationships can be seen as edges
- We consider three database models that come close to ER
- 1. Neo4j
 - Comes close; lacks inheritance
 - Concept of weak entities is not articulated
- 2. Object-relational model and ODMG (Chapter 9)
 - ODMG stands for Object Data Management Group
 - “ODMG” also stands for the standard recommended by ODMG
 - Edges are seen as directed, e.g., Employee WorksIn Dept
Syntactically, if e is an employee object, e.WorksIn is the Dept object
 - Edges can be reversed: d.HasEmployees
 - Problem: no industrial object-oriented database product available
One serious candidate O2 – existed once is no more
- 3. OOXML and XQuery
 - OOXML / OOXQuery are our o-o dialects of XML / XQuery
 - OOXML helps express directed edges and inheritance via dotted expressions
 - Object-orientation and dotted expressions can be made available to all XML technologies, including XQuery

- Weak entities are absorbed via nesting in XML, hence in OOXML
- This is perhaps the only (almost) working o-o “product” today
It seems (almost) ODMG compliant