

CS 4221: Database Design

Physical Database Design

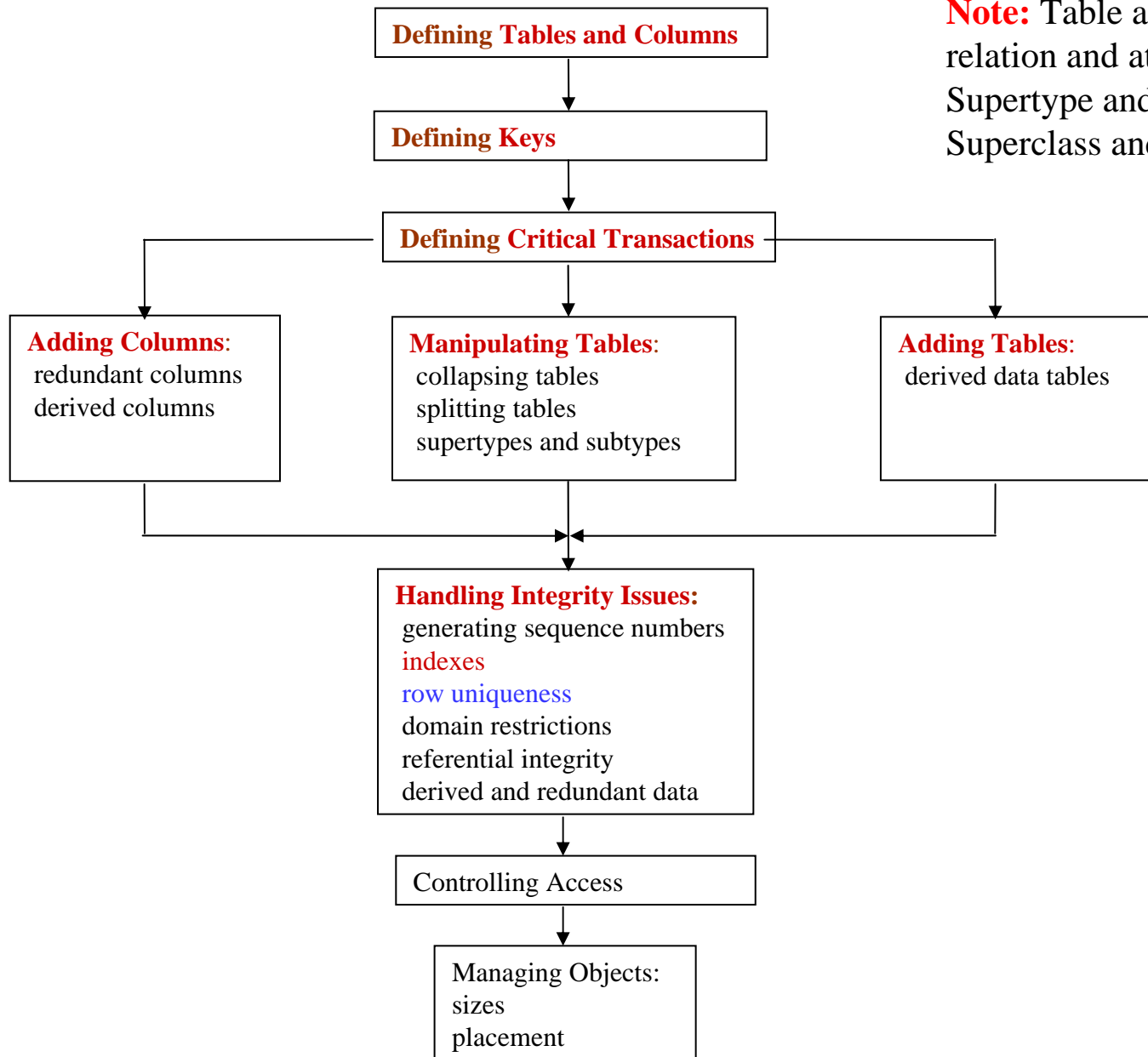
Ling Tok Wang
National University of Singapore

Physical Database Design

- It is the process of transforming a **logical data model** into a **physical model** of a database.
- Unlike a logical design, a physical database design is **optimized** for **data-access paths, performance requirements** and **other constraints** of the target environment, i.e. hardware and software.
- Before you can begin the physical design, you must have:
 - (1) **logical database design**
 - minimally third normal form
 - (2) **Transaction characterization**, such as
 - most frequent transactions
 - most complex or resource-insensitive transactions
 - distributions of transactions, over time
 - mix of insert, update, delete and select statements
 - most critical transactions to the applications
 - (3) **Performance requirements**

Ref: R Gillette, D Muench, and J Tabaka. Physical Database Design for SYBASE SQL Server. Prentice Hall, 1995.

Physical database design activities



Note: Table and column mean relation and attribute. Also Supertype and Subtype mean Superclass and Subclass, resp.

- 1. Defining Tables and Columns** – The initial transformation of the logical model into a physical model, including naming objects, choosing data types and lengths, and handling null values.
- 2. Defining Keys** – Choosing primary and foreign keys, including the use of surrogate keys.
- 3. Identifying Critical Transactions** – Identifying business transactions that are high-value, mission-critical, frequently performed, or costly in terms of computing resources.
- 4. Adding Redundant Columns** – The first of a series of denormalization techniques: adding columns to tables that exist in other tables.
- 5. Adding Derived Columns** – Adding a column to a table based on the values or existence of values in other columns in any table.
- 6. Collapsing Tables** – Combining two or more tables into one table.
- 7. Splitting Tables** – Partitioning a table into two or more disjoint tables. Partitioning may be horizontal (row-wise) or vertical (column-wise).

8. **Handling Supertypes and Subtypes** – Deciding how to implement tables that are involved in a supertype-subtype relationship in the logical model.
9. **Duplicating Parts of Tables** – Duplicating data vertically and / or horizontally into new tables.
10. **Adding Tables for Derived Data** – Creating new tables that hold data derived in columns from other tables.
11. **Handling Vector Data** – Deciding how to implement tables that contain plural attributes or vector data. Row-wise and column-wise implementations are discussed.
12. **Generating Sequence Numbers** -- Choosing a strategy to generate sequence numbers, and the appropriate tables and columns to support the strategy.
13. **Specifying Indexes** – Specifying indexes to improve data access performance or to enforce uniqueness.
14. **Maintaining Row Uniqueness** – Maintaining the uniqueness or primary-key values.

15. **Handling Domain Restriction** – Defining SQL Server rules and defaults on the columns of a table to maintain valid data values in columns.
16. **Handling Referential Integrity** – Deciding how to handle primary-key updates and deletes, and foreign-key inserts and updates. Using triggers to ensure referential integrity.
17. **Maintaining Derived and Redundant Data** – Specifying how data integrity will be maintained if the data model contains derived or redundant data.
18. **Handling Complex Integrity Constraints** – Deciding how to handle complex business rules such as sequence rules, cross-domain business rules, and complex data domain rules. Using triggers to implement complex business rules.
19. **Controlling Access to Data** – Restricting access to commands and data.
20. **Managing Object Sizes** - Calculating the estimated size of a database and its objects.
21. **Recommending Object Placement** – Allocating databases and their objects on available hardware to achieve optimal performance.

Physical database design goals

- improve system performance
 - reduce disk I/O
 - reduce joins
- embed business rules into the database design
 - through defaults, rules, constraints, stored procedures, or triggers
- make it understandable to users
 - use meaningful and indicative names for tables and columns

1. Defining Keys

- If there are more than one candidate key in a table, select the **primary key** as below:
 - select the key which transactions will know about most often. This will avoid additional lookups.
 - select the shortest length key when used in indexes
 - consider what other keys are available in other tables on which to join.
 - criteria for primary selection as mentioned in our tutorial.
- **Surrogate keys** are columns with no business meaning that are added to tables to represent one or more existing columns.
- Surrogate key does not replace the logical primary key; instead **it redefine the primary key for use as a foreign key in other tables**. It is for efficiency purpose.

E.g. The title-id column of the Titles table is a surrogate-key value and replaces the title, pubname, and pub-date fields as the primary key.

Titles (title-id, title, type, pubname, price, pub-date, ...)

- **Large keys** can have a significant effect on overall system performance.
- **Surrogate-key candidates** include:
 - tables that with very large or multi-column primary keys
 - text columns that require indexing
- **Benefits** of a smaller surrogate key
 - easier to write SQL code to join table
 - reduce the size of the tables with use it as a foreign key
 - decrease the size of foreign-key indexes
 - increase performance on queries accessing tables with surrogate key values

2. Identifying Critical Transactions

To understand the transactions and performance requirements, you need to know:

- **types of transactions** (select, insert, update, delete)
- tables and column affected by each transaction
- select criteria – fixed or variable (i.e. pre-defined queries or ad-hoc queries)
- **frequency** and **volume** of each transaction
- how many rows (percentage) are typically affected (select or modified)
- size (no. of rows and total bytes) of tables involved
- **when** the transaction is executed
 - during the day or after office hours
- **relative importance** of each transaction
 - who use it, how often, how critical is it to the business process
- **response time** or throughput desired
- security and integrity
- how many tables will be **joined**
- **sort order**

- Identifying transactions **unlikely** to meet performance requirements:

These **critical transactions** usually are:

- most frequently performed transactions
- transactions performed by key personnel
- transactions affecting many rows
- resource-intensive transactions
- mission critical transactions
- high volume transactions

Attention must be paid to the distribution of transactions with respect to **time**, if this is not uniform (e.g. peak periods and specific run times).

3. Adding Redundant Columns

- required when an unaccepted number of **joins** is needed to perform a critical transaction.
- add redundant columns in order to **reduce the no. of joins**.
 - It is a **de-normalization process**. Tables will not be in 3NF.
- The concept of **strong FD, weak FD, relax-replicated 3NF relation** can be used as the theory for this process.
- **Benefits:**
 - better response time
 - The chance to eliminate a foreign key
 - The reduction of lock contention; this cut down blocking or deadlock situations.

- You should be aware that:
 - The modified table will grow in size.
 - The larger no. of data pages will slow performance of queries not benefiting from elimination of the joins; as the no. of I/Os required to process the table is greater.
 - The duplicated column data will require maintenance.

- **Example**

publisher (pub-id, pubname, city, state)

Titles (title-id, title, type, pub-id, price, pubname, ...)

- pubname is duplicated in Titles table

(see previous notes on relax-replicated 3NF)

4. Adding Derived Columns

- When you expect that the performance requirements for a **critical transactions** will not be met because of a costly, recurring calculation based on relative static data, then add derived column will help.
- Derived data may include:
 - column data aggregated with **SQL aggregate function** such as **sum()**, **avg()**, over N detail rows
 - column data which is calculated using formulas over N rows.
 - counts of details rows matching specific criteria

Example: Total-sales in Titles table

Titles (title-id, title, type, pub-id, price, **total_sales**,
pubdate, pubname)

5. Collapsing Tables

- Required when the application program must frequently access data in **multiple tables** in a single query.

e.g. Combining the publishers and Titles tables will improve the performance of the critical query

Titles (title-id, title, type, pub-id, price,
total-sales, **pubname**, **city**, **state**, pubdate)

- de-normalization
- similar to adding redundant columns
- in order to get better performance
- a research area:
materialized database

6. Splitting Tables

- Required when it is more advantageous to access a **subset** of data, and no important transactions rely on a consolidated view of the data.

- **Vertical table splits:**

e.g. Emp (Eno, name, salary, tax, mgr#, dept#)

can be split to 2 tables:

Emp_bio (Eno, name, mgr#, dept#)

Emp_comp (Eno, salary, tax)

- The **rows are smaller**. This allows more rows to be stored on each data page, therefore no. of I/Os is reduced.
- Each **fragment** (smaller table) must include the **primary key** of the original table.

- **Horizontal table splits**

e.g. You can form horizontal fragments of the Supplier table based on values of the city column

Supplier (sno, sname, city, status)

Supplier_boston (sno, sname, status)

Benefits:

- A table is large and reducing its size **reduces** the no. of index pages read in a query
- The table split corresponds to an actual physical separation of the data rows, as in different **geographical sites**.
- Table splitting achieves specific distribution of data on the available physical media
- To achieve **domain key normal form**.

7. Handling Supertypes and Subtypes

- Decide how to involved in a supertype/subtype relationship in the logical data model. (i.e. superclass subclass isa relationship)

Example From logical database design

employee (eno, name, salary, tax, mgr#, dept#)
contractor (eno, billing-rate, contracting-title)
consultant (eno, billing-rate, consulting-title, mentor)
regular-staff (eno, prof_soc_num)

Also

contractor [eno] isa employee [eno]
consultant [eno] isa employee [eno]
regular-staff [eno] isa employee [eno]

They are the 3 supertype/subtype relationships (see Fig 8.1).

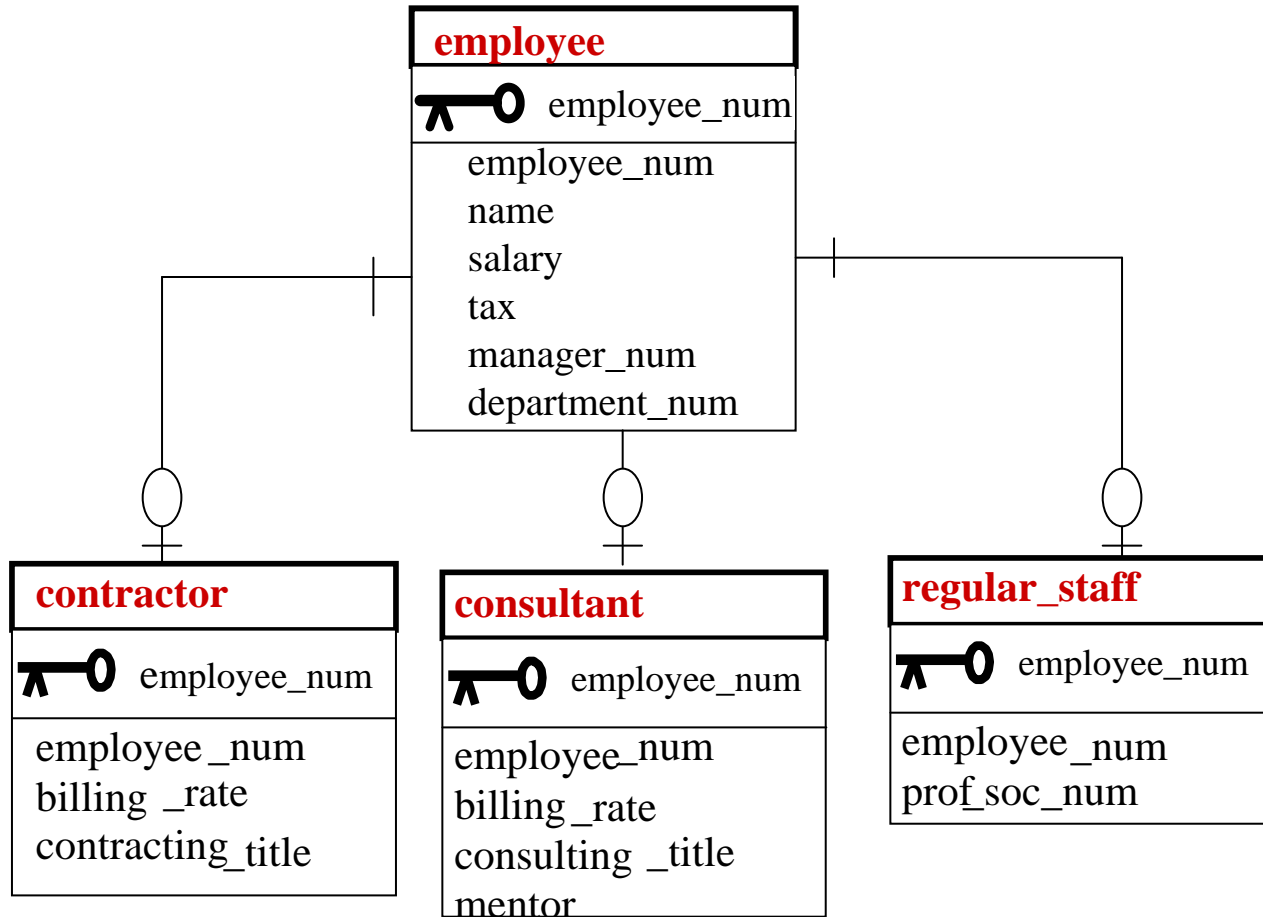


Fig 8.1 Logical design starting point

- There are 3 common physical design scenarios for subtype-supertype relationships.
 - (1) one supertype table and multiple subtype tables
 - (2) one supertype table only
 - (3) multiple subtype tables only

(1) Single supertype and multiple subtype tables

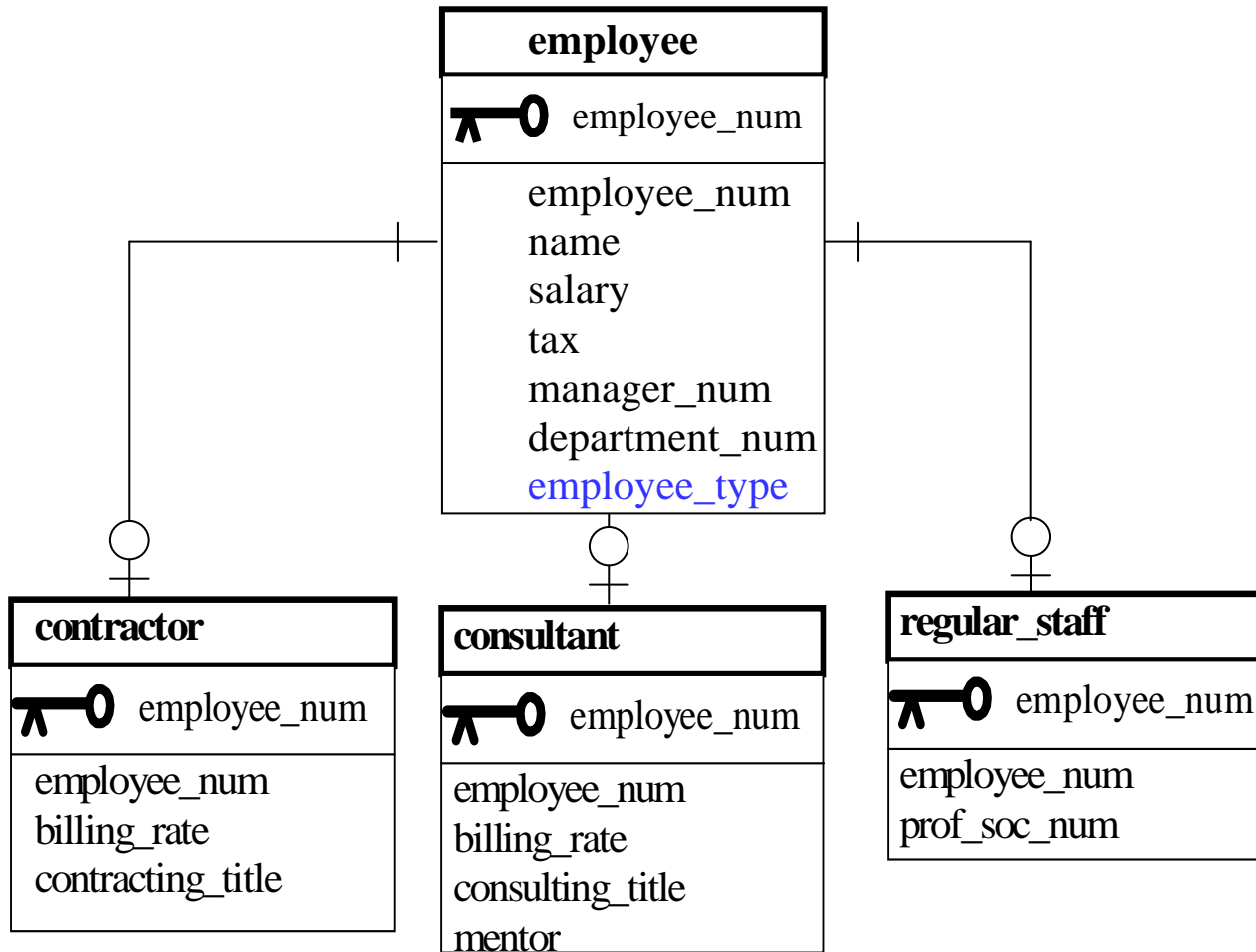



Fig 8.2 Single supertype/multiple subtype example

- The supertype table employee has an **additional attribute**, i.e., **employee_type**, which determines the appropriate sub-table for the employee.
- use this technique when the subtypes have many differences and **few common attributes** and **reports rarely** require the supertype data and subtype data.
- It is useful when the **no. of subtypes is initially unknown**

(2) Single Supertype table only

Fig 8.3 Single supertype table example

employee	
	employee_num
<hr/>	
	employee_num
	name
	salary
	tax
	manager_num
	department_num
	consulting_title
	contracting_title
	billing_rate
	mentor
	prof_soc_num
	employee_type

- This technique is appropriate if the subtypes
 - have similar columns
 - are involved in similar relationships
 - are frequently accessed together
 - are infrequently accessed separately
- All columns of all the subtype tables are included (with **many null values**) in the supertype table
An extra attribute **employee_type** is included which determines the subtype of employee rows.
- **many null values**

(3) Multiple Subtype Tables Only

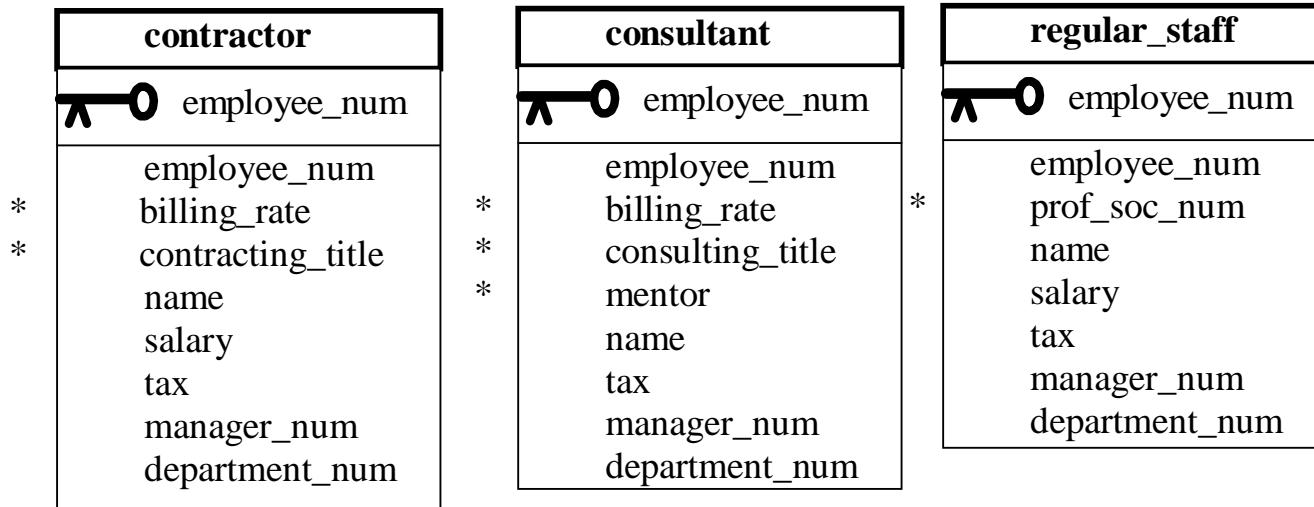


Fig 8.4 Subtype tables only example.

* indicates the attribute is a subtype entity's attribute

- Use it if a supertype entity in the logical database design existed only to **clarify concepts or to add clarity** to the model, or if the supertype entity has **no attributes other than the primary key**
- The **union** of all subtype tables will return data on all employees.

8. Adding Tables for Derived Data

- Required when the structure of the database does **not** support commonly accessed information, and the derived data does **not** naturally fit on an exist table.
- many applications or reports call for data **summaries**, often at more than one level of **grouping** for the same source data.
- generating summaries with large tables, may become a performance bottleneck.

Example Summary table

Titles (title-id, title, type, pub-id, price, pubdate)

Summary-table (type, total-sales)

- The total-sales attribute stores the total sales for the same type of books
- **Triggers** are required to **update** the summary-table.

9. Specifying Indexes

- Indexes can be used to **improve data access performance**, to **enforce uniqueness**, or to **control data distribution**.
- Indexes may be **clustered** or **non clustered**, **unique** or **non unique**, or **concatenated**.
- Testing and trial-and-error during production may indicate other index choices.
- A table's indexes must be maintained with every insert, update, and delete operation performed on the table.
- Be careful **not to over index**.
Incorrect index selection can adversely affect the performance.
- The greatest problem will be deriving the best set of indexes for the database when conflicting applications exist (i.e. applications whose access needs and priorities are in conflict).

- You may need to **split up** or **duplicate** a database into another database in order to support equally critical but opposing indexing strategies, particularly with request to the **clustered index**, where only **one** is allowed per table.
- **Index density** = $1 / \text{total no. of unique values}$
 - e.g. If there are 20 colors for cars then the index density for colors is $1/20 = 0.05$.
 - e.g. The index density of the **primary key** of a table is $1/\text{no_of_row}$
- **Selectivity** = Index density * total no. of rows
 - The **more selective** (**lower selectivity value**) this number is, the more likely the SQL query optimizers will choose to use the index since it can assume fewer rows will be required to answer the query.
 - E.g.** If there are 200 values and 400 rows, then the selectivity value is $1/200 * 400 = 2$, indicating that on average only 2 rows should be returned for each index value.
 - **The selectivity for primary key is 1.** Q: Why? The most selectivity?
 - With a **composite index**, the density should get **lower** for each additional column specified in the index, hence making the index more selective.

- Data-value uniqueness
 - Indexes can enforce uniqueness of data values in the column on which they are placed.
 - With Sybase SQL server 10.x, you can use **primary key** or **unique** constraints to enforce uniqueness. Both of these constraints will cause an index to be formed for the named columns.

- **Clustered indexes**

- an index in which the physical order of rows and the logical (indexed) order are the same. The leaf level of a clustered index represents the data pages themselves.

- Only one clustered index is allowed for each table. Usually, the primary key is the clustered index on the tables, but not always.

Instead you may want to choose the attribute which is used to specify a range in a where clause.

- Clustered indexes are implemented as B-trees in SQL servers. Insertions may cause the splitting of the leaf nodes of a B-tree.

- **Non clustered indexes**

- an index which maintains a **logical ordering** of data rows without altering the **physical ordering** of the rows.
- **Foreign keys** are good candidates for non cluster indexes.
- Non clustered indexes are implemented by B-trees.
A **B-tree of pointers** (to the rows in the table) is maintained for the indexed column values in a **sorted order**, even though the data rows themselves are not physically ordered according to the column values.

E.g. Emp (eno, name, mgr#, dept#)

We can define a non clustered index for dept# of the table Emp.

Note: dept# is a foreign key in Emp table.

- Tables that should be considered as candidates for indexes are:
 - tables that are used in critical transactions and that have a set of search criteria (or limit ranges)
 - tables involved in multi-table joins
 - tables with a large no. of rows
 - tables that require enforcement of uniqueness

- Identifying Columns for Indexes
 - columns used to specify *range* in the *where* clause (clustered index)
 - columns used to *join* one or more tables, usually *primary* and *foreign keys*
 - columns likely to be used as *search arguments*
 - columns used to match an *equi-join query*
 - columns used in *aggregate functions*
 - columns used in a *group by* clause
 - columns used in an *order by* clause

- In environment where **deletes and inserts are frequent**, such as many real-time transaction processing applications, you may **avoid the clustered index**.
- **Notes:** An index with **low selectivity** (higher value) will never be used by the **query optimizer**.

E.g. **Sex** attribute in person table has index density = $\frac{1}{2}$, and selectivity is equal to half of the no of rows of the table (very high value). It has very low selectivity. Should not index it. Query optimizer will not use its index.

- The optimizer will not use a clustered index of a table with very **few rows** (e.g. tables with less than 3 to 5 data pages total).