System R
cs262a, Lecture 2

Ali Ghodsi and Ion Stoica

(adapted from Joe Hellerstein's notes)

Databases

Store two types of information. What are they?

» Contents of records

» How records are connected together.

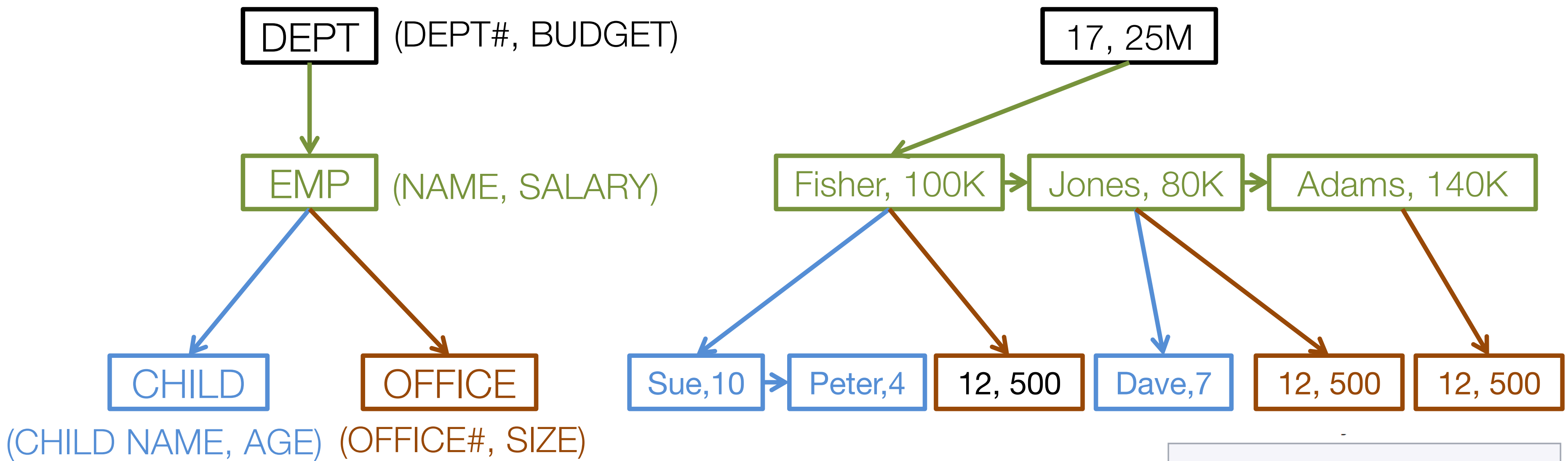
How do you search a database?

» You specify detailed algorithms that traverse the connections to get the answer.

Examples: search/insert/delete in linked lists, trees, etc

Before relational DBs: hierarchical and network

Hierarchical Model*



1966: IMS (IBM Management System)

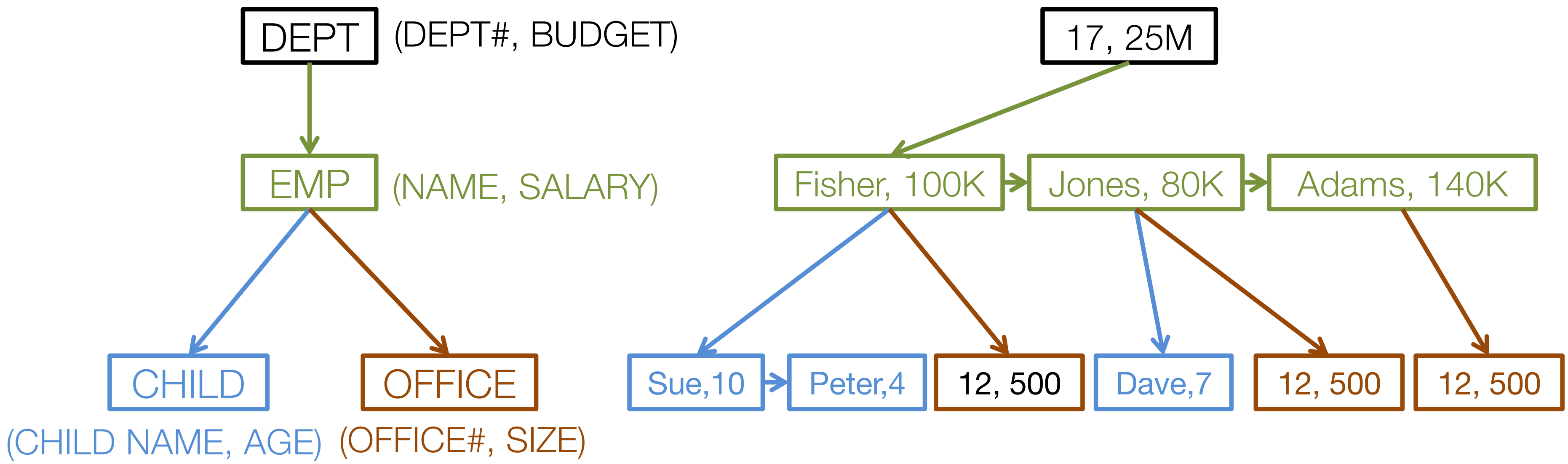
» Designed for Apollo program for managing inventory for Saturn V and space vehicle

» Still in use today!

Initial release	1966; 52 years ago
Stable release	IMS V15 / October 3, 2017; 3 months ago
Development status	Active
Operating system	z/OS V2.2 or later
Platform	IBM System z
Type	Database & transaction processing subsystem
License	proprietary
Website	IBM IMS Product Page

*examples from "Network hierarchies and relations in database management systems" by M. Stonebraker and G. Held

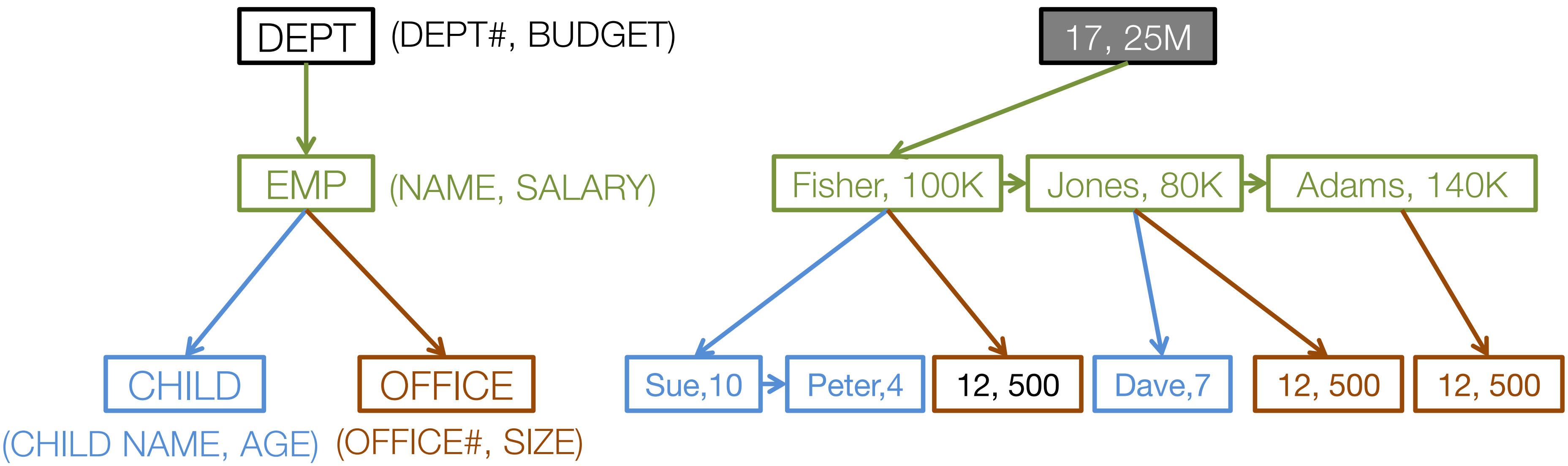
Hierarchical Model*




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return "no such department"
      FIND 1st SON OF CURRENT RECORD
          if failure; return "no employee in this department"
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

*examples from "Network hierarchies and relations in database management systems" by M. Stonebraker and G. Held

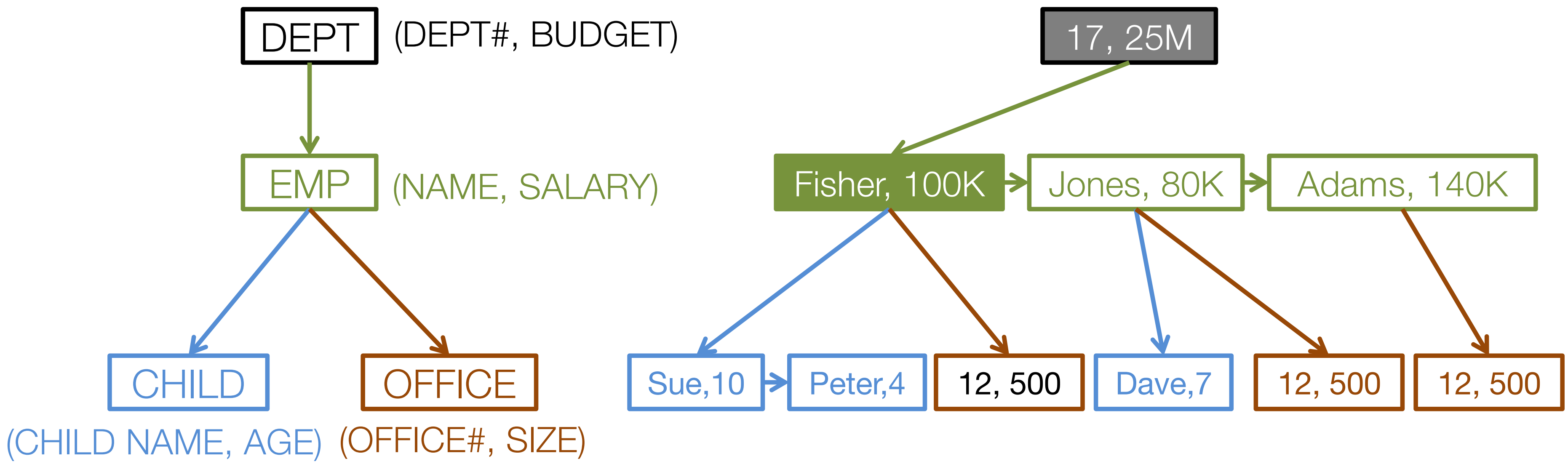
Hierarchical Model




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return “no such department”
      FIND 1st SON OF CURRENT RECORD
          if failure; return “no employee in this department”
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Output:

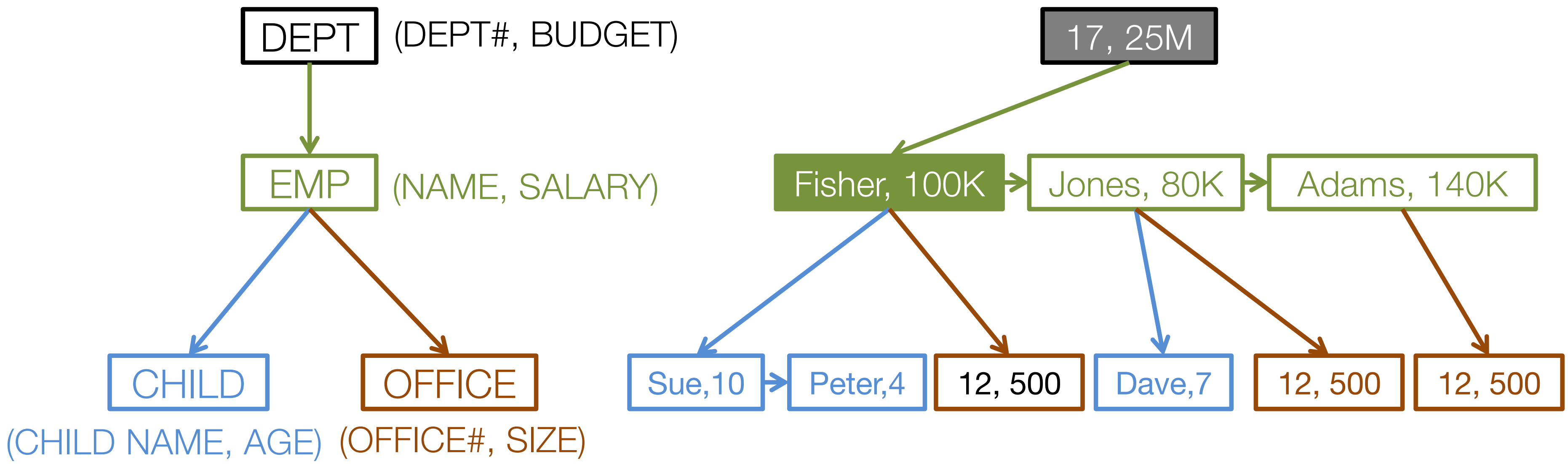
Hierarchical Model




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return "no such department"
      FIND 1st SON OF CURRENT RECORD
          if failure; return "no employee in this department"
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Output:

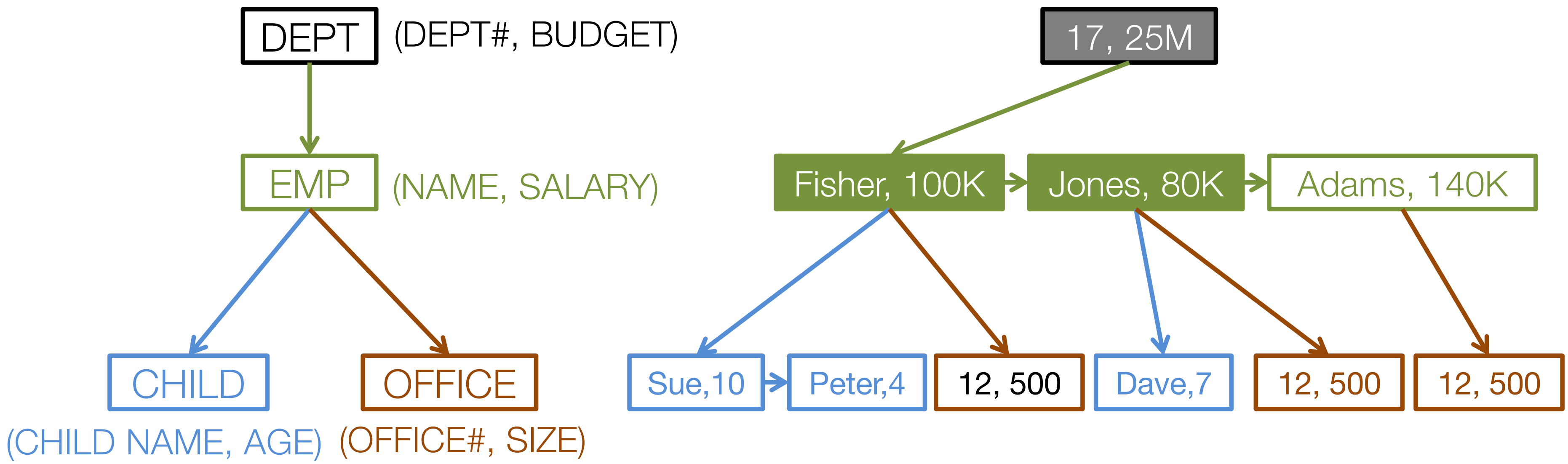
Hierarchical Model




```
# find names of all employees in department 17
        FIND DEPT RECORD WHERE DEPT# = 17
             if failure; return “no such department”
        FIND 1st SON OF CURRENT RECORD
             if failure; return “no employee in this department”
LOOP         save name
        FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
        GO TO LOOP
```

Output: Fisher

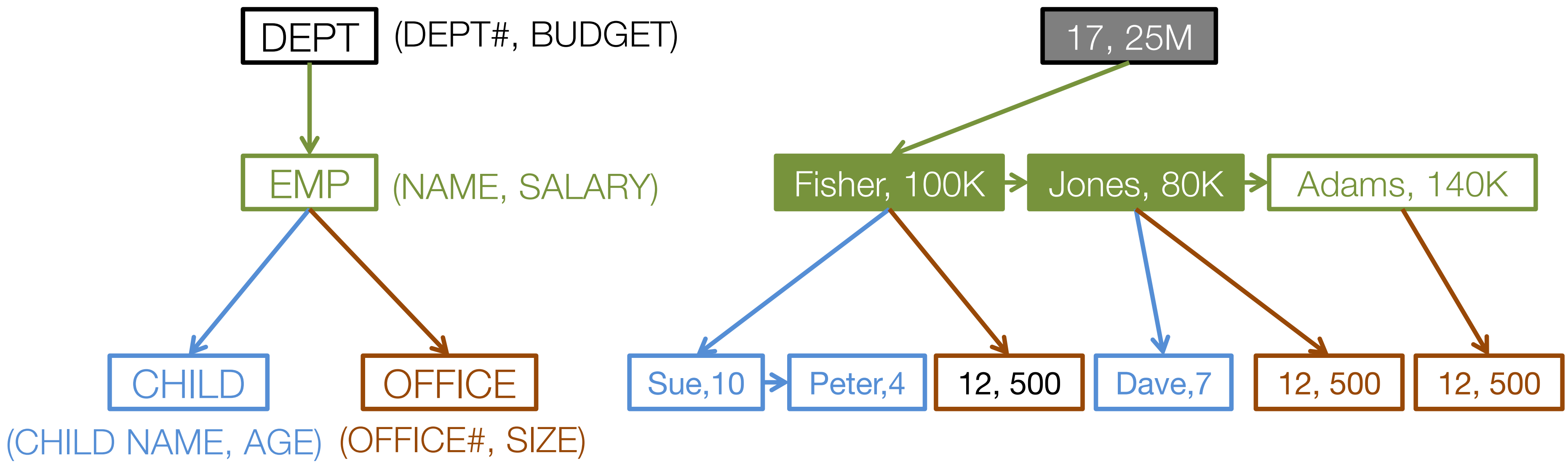
Hierarchical Model




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return “no such department”
      FIND 1st SON OF CURRENT RECORD
          if failure; return “no employee in this department”
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Output: Fisher

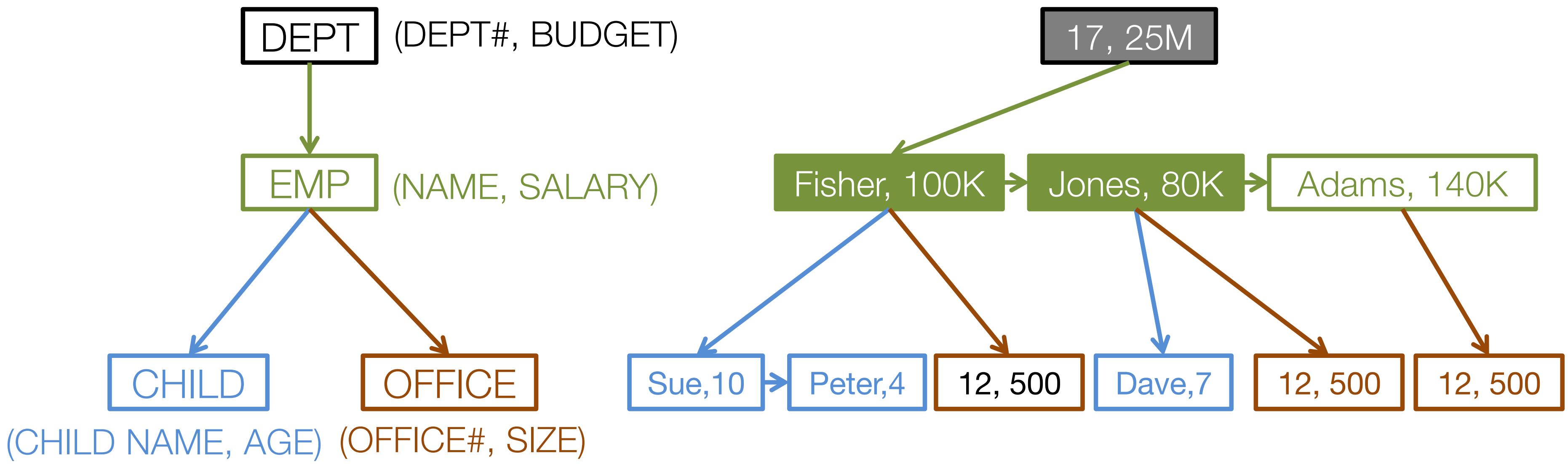
Hierarchical Model




```
# find names of all employees in department 17
        FIND DEPT RECORD WHERE DEPT# = 17
             if failure; return “no such department”
        FIND 1st SON OF CURRENT RECORD
             if failure; return “no employee in this department”
LOOP         save name
        FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
        GO TO LOOP
```

Output: Fisher

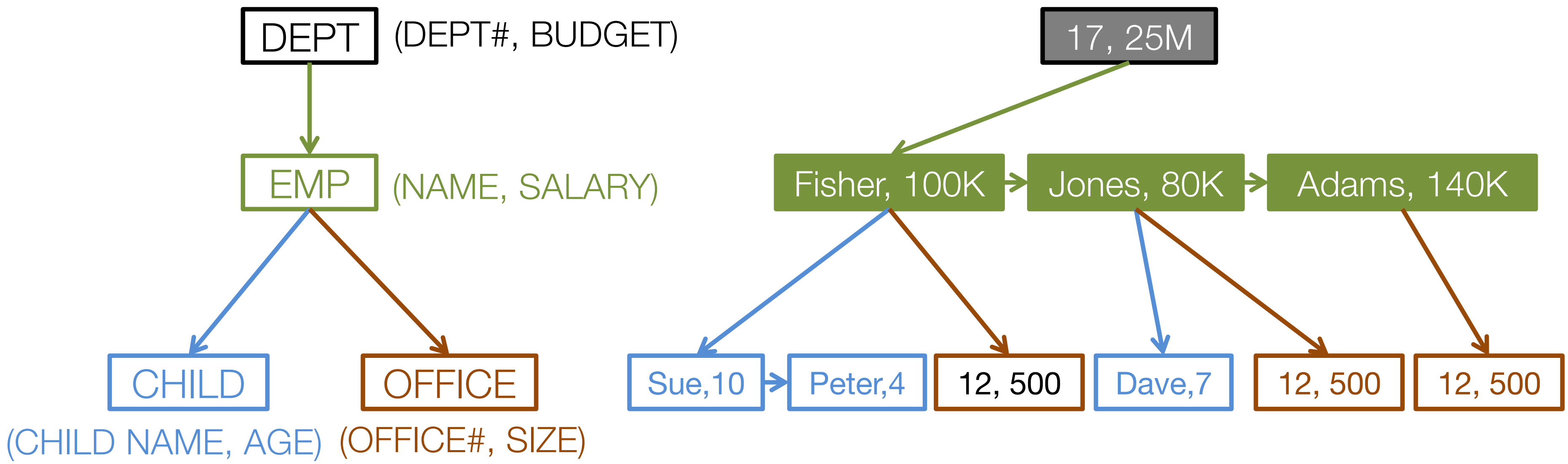
Hierarchical Model




```
# find names of all employees in department 17
        FIND DEPT RECORD WHERE DEPT# = 17
            if failure; return "no such department"
        FIND 1st SON OF CURRENT RECORD
            if failure; return "no employee in this department"
LOOP        save name
        FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
        GO TO LOOP
```

Output: Fisher Jones

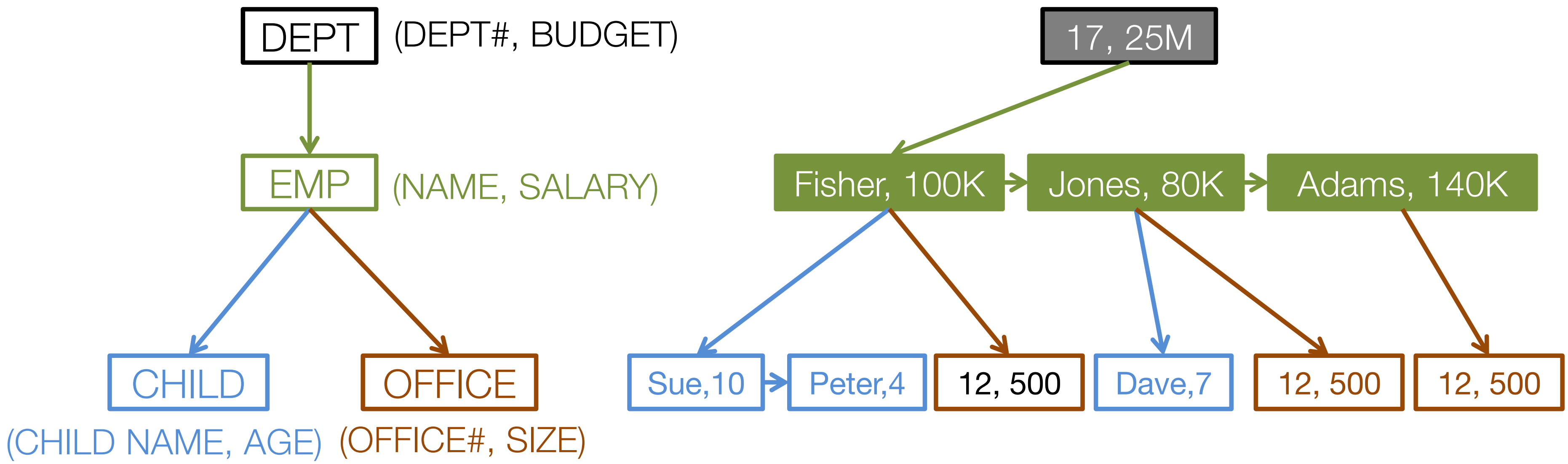
Hierarchical Model




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return "no such department"
      FIND 1st SON OF CURRENT RECORD
          if failure; return "no employee in this department"
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Output: Fisher Jones

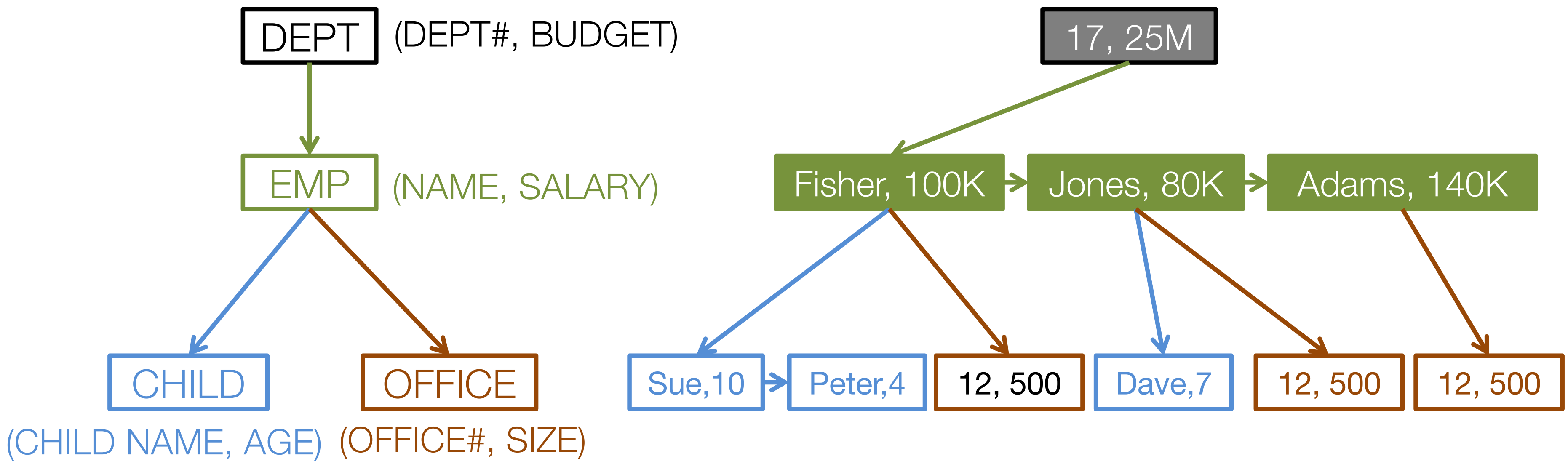
Hierarchical Model




```
# find names of all employees in department 17
        FIND DEPT RECORD WHERE DEPT# = 17
            if failure; return “no such department”
        FIND 1st SON OF CURRENT RECORD
            if failure; return “no employee in this department”
LOOP        save name
        FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
        GO TO LOOP
```

Output: Fisher Jones

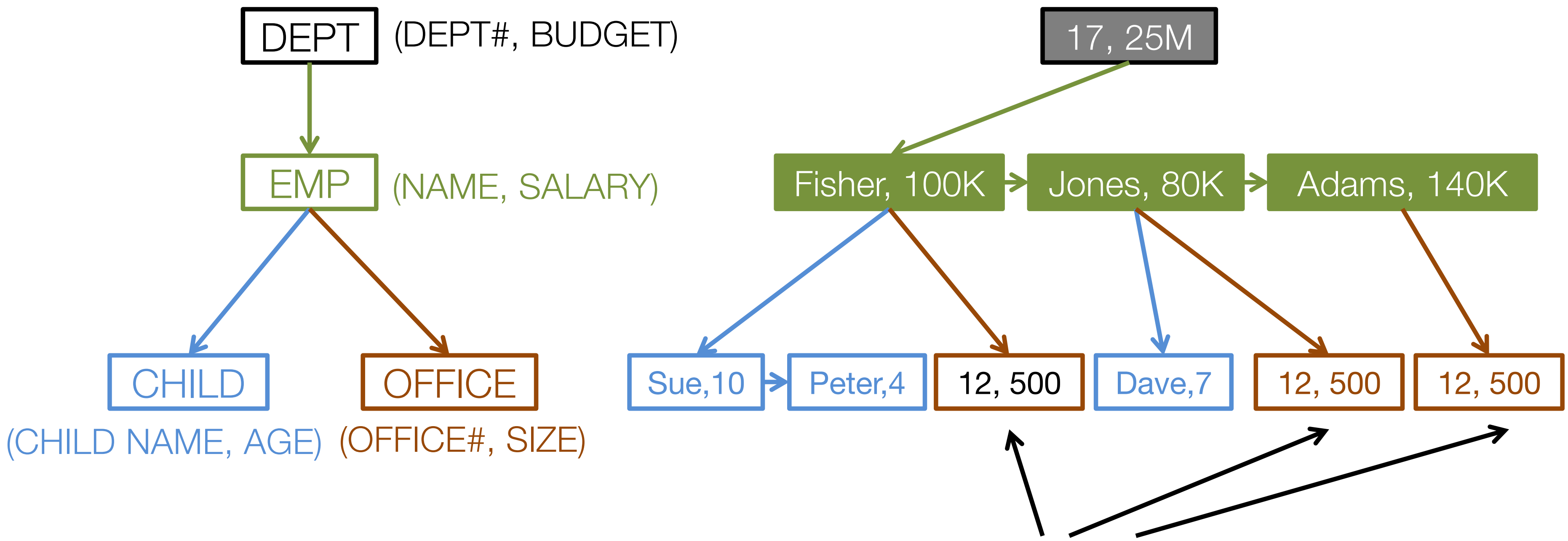
Hierarchical Model




```
# find names of all employees in department 17
        FIND DEPT RECORD WHERE DEPT# = 17
             if failure; return "no such department"
        FIND 1st SON OF CURRENT RECORD
             if failure; return "no employee in this department"
LOOP         save name
        FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
        GO TO LOOP
```

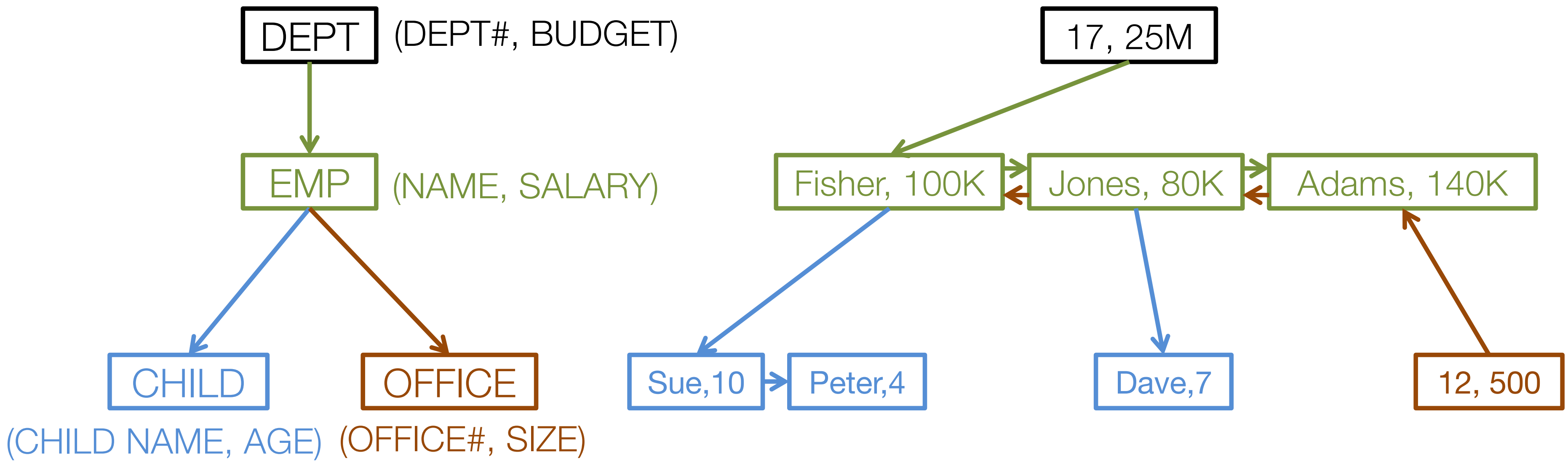
Output: Fisher Jones Adams

Hierarchical Model: Challenges



1) Duplicate records
2) Requirements to have a parent; deletion anomalies

Network Model

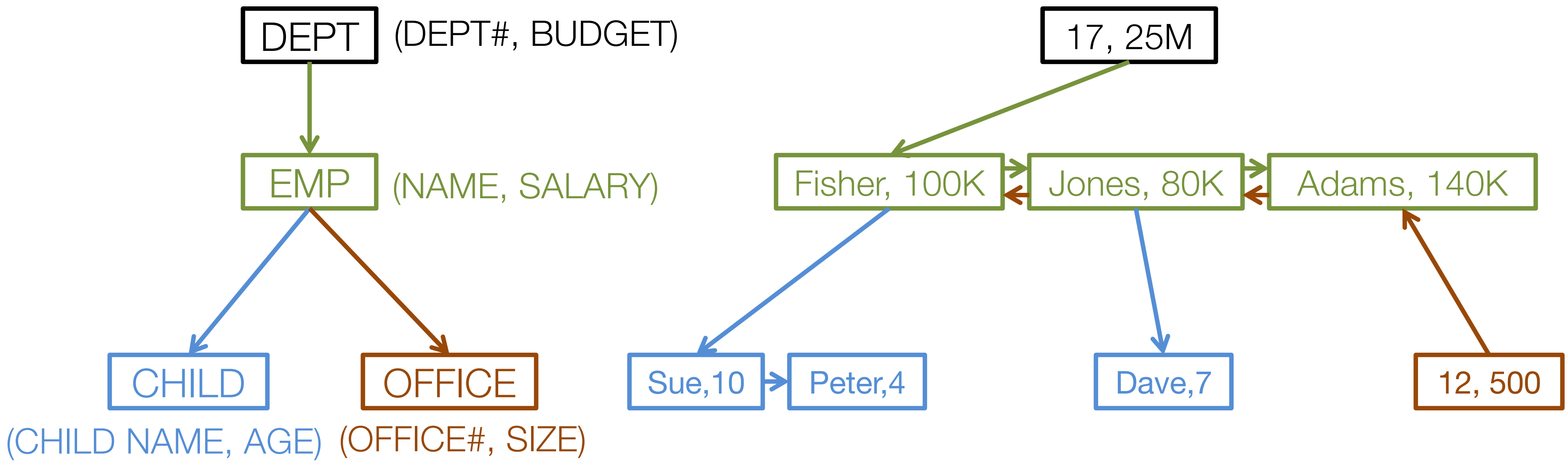


CODASYL (Conference/Committee on Data Systems Languages)

» 1969: CODASYL data model
Designed by **Charles Bachman**, Turing Award, 1973

» Also led to development of COBOL

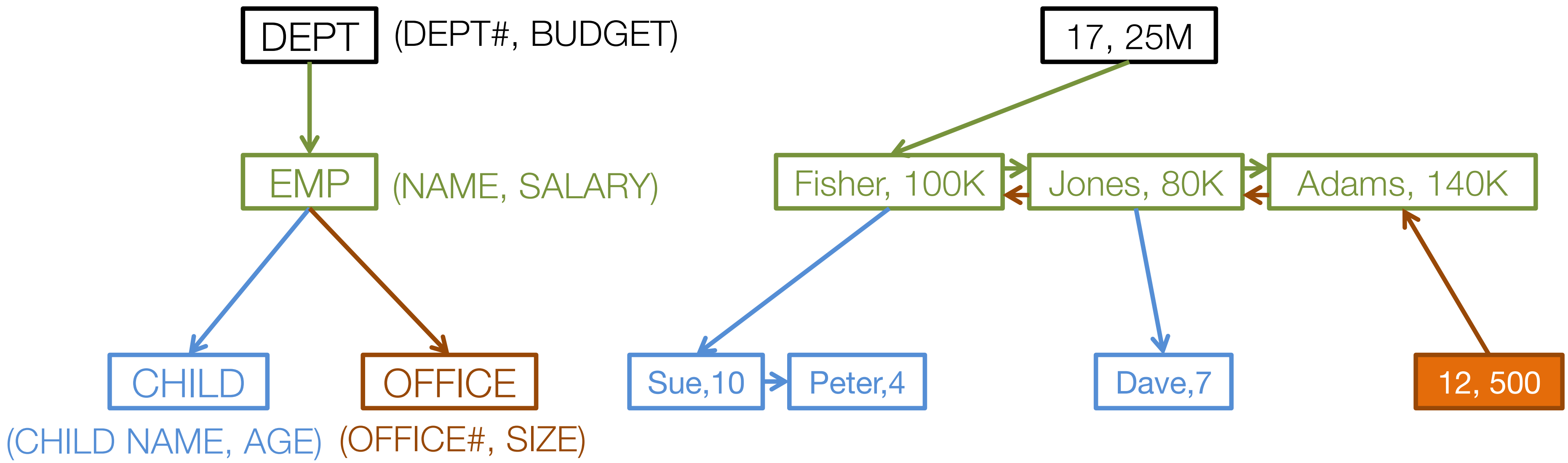
Network Model*




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

*examples from "[Network hierarchies and relations in database management systems]" by M. Stonebraker and G. Held

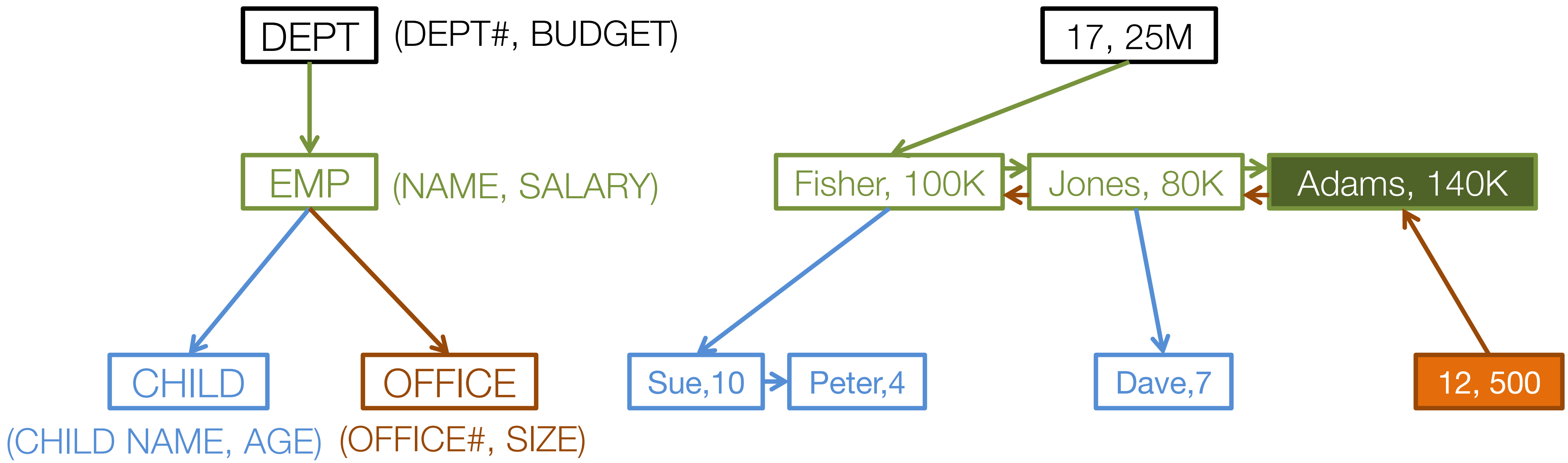
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output:

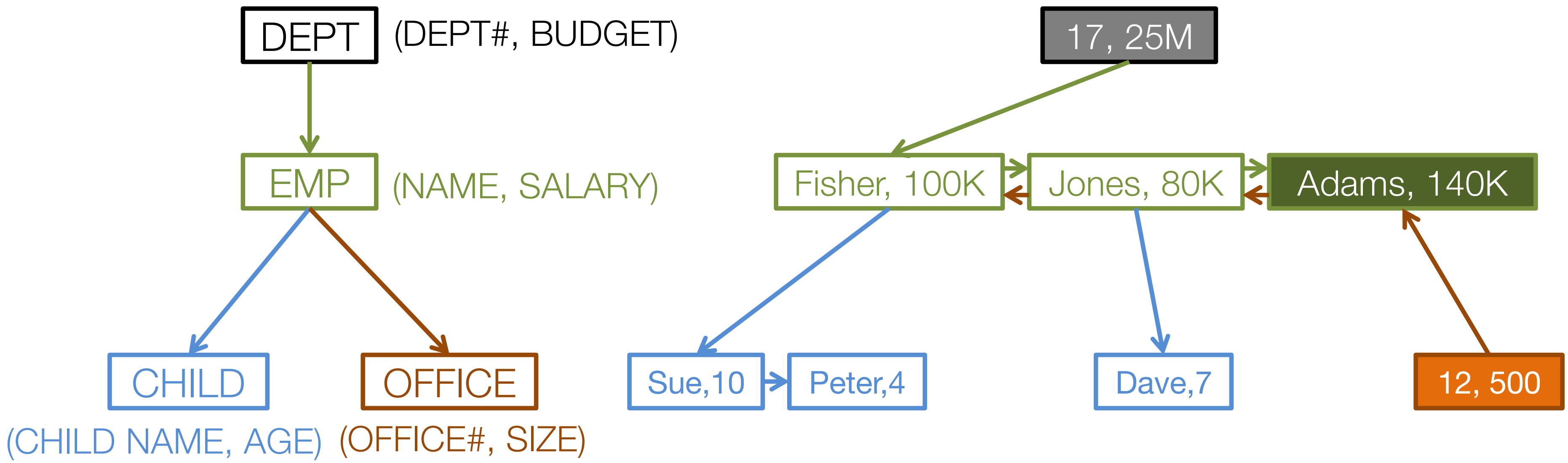
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output:

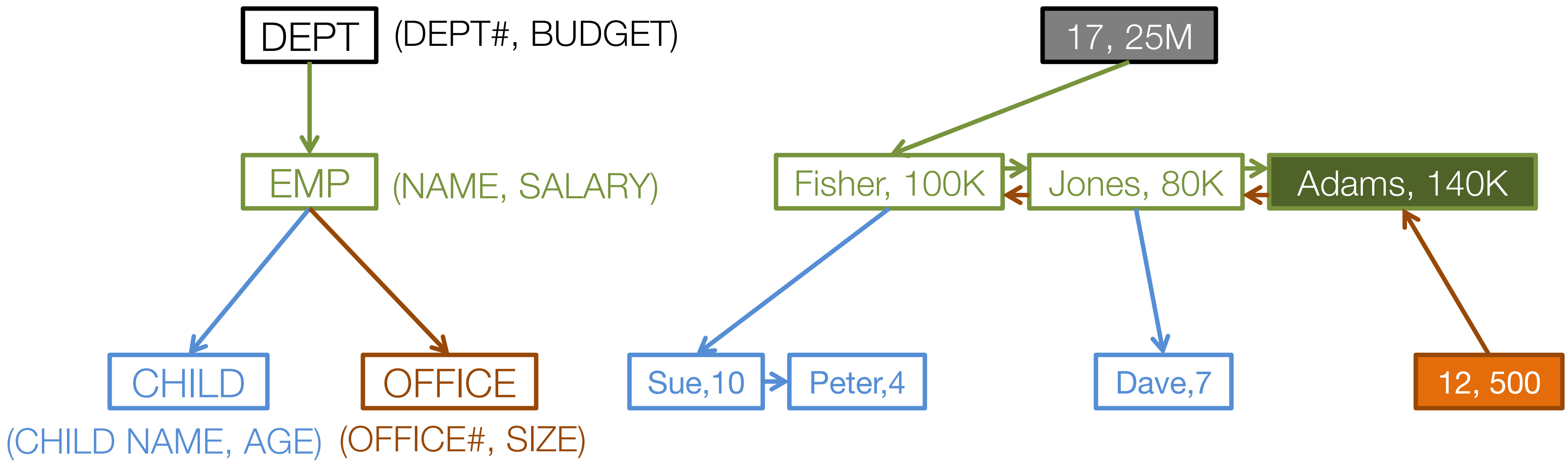
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output:

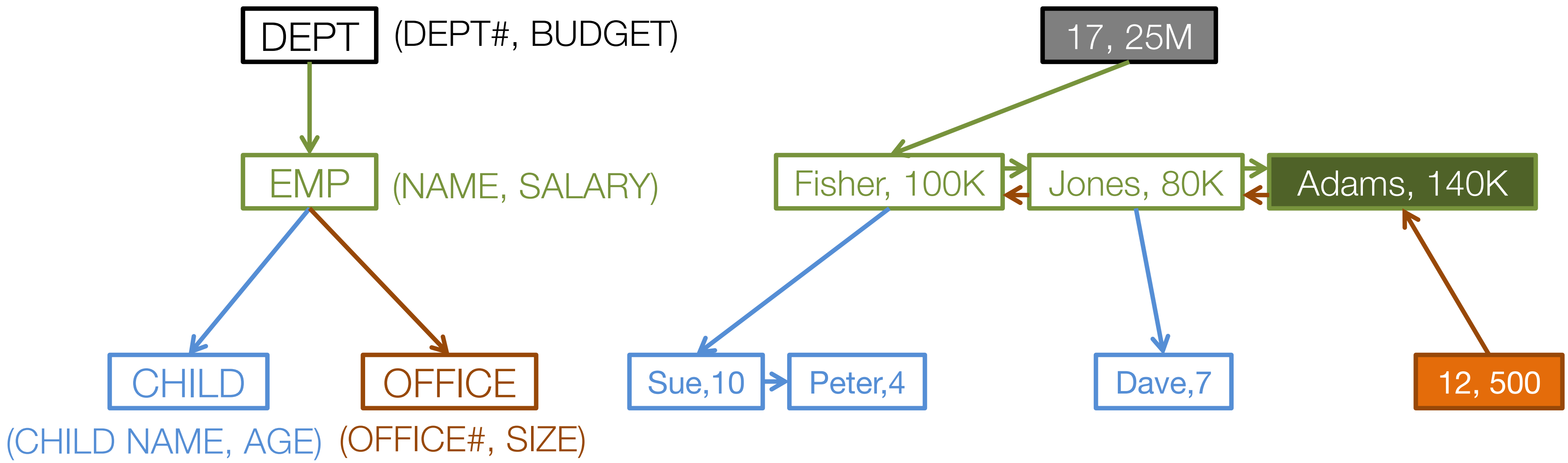
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

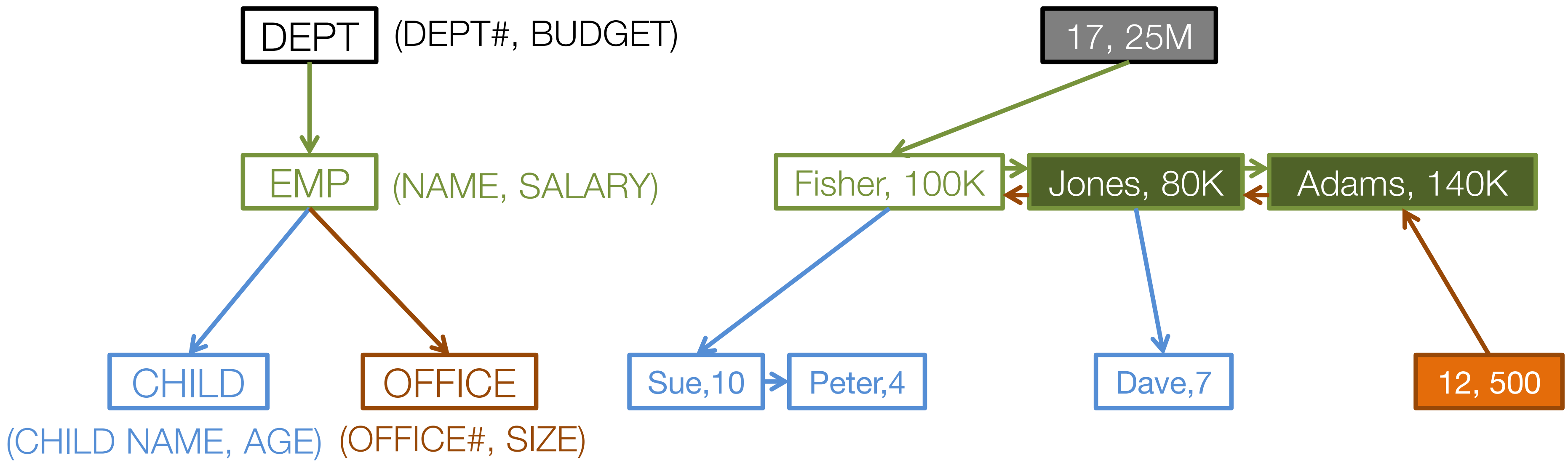
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

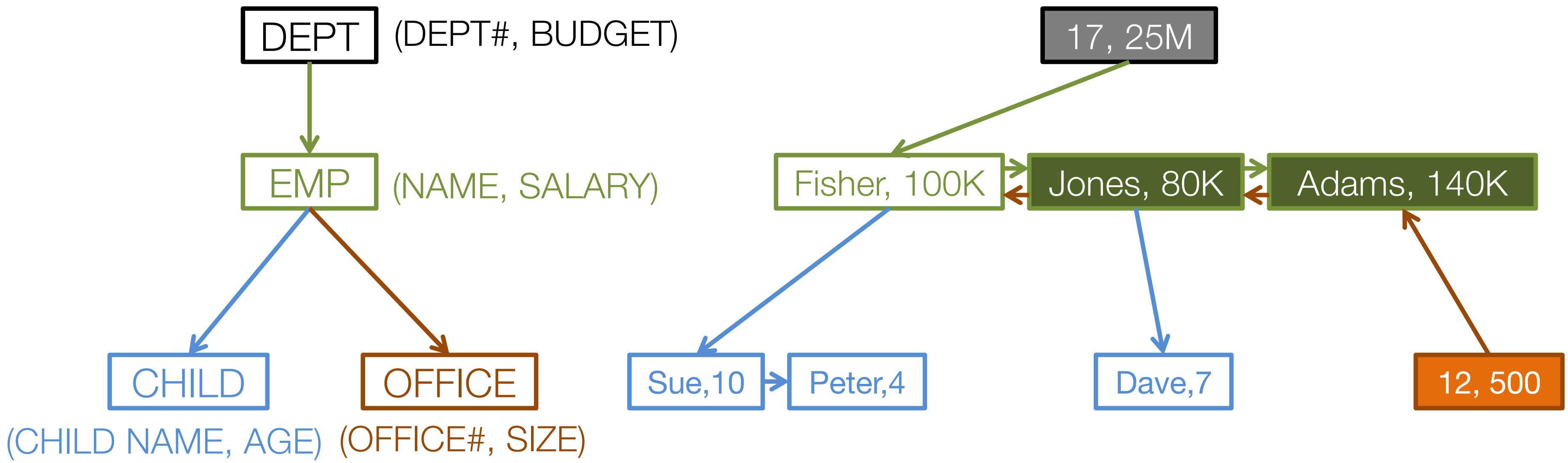
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

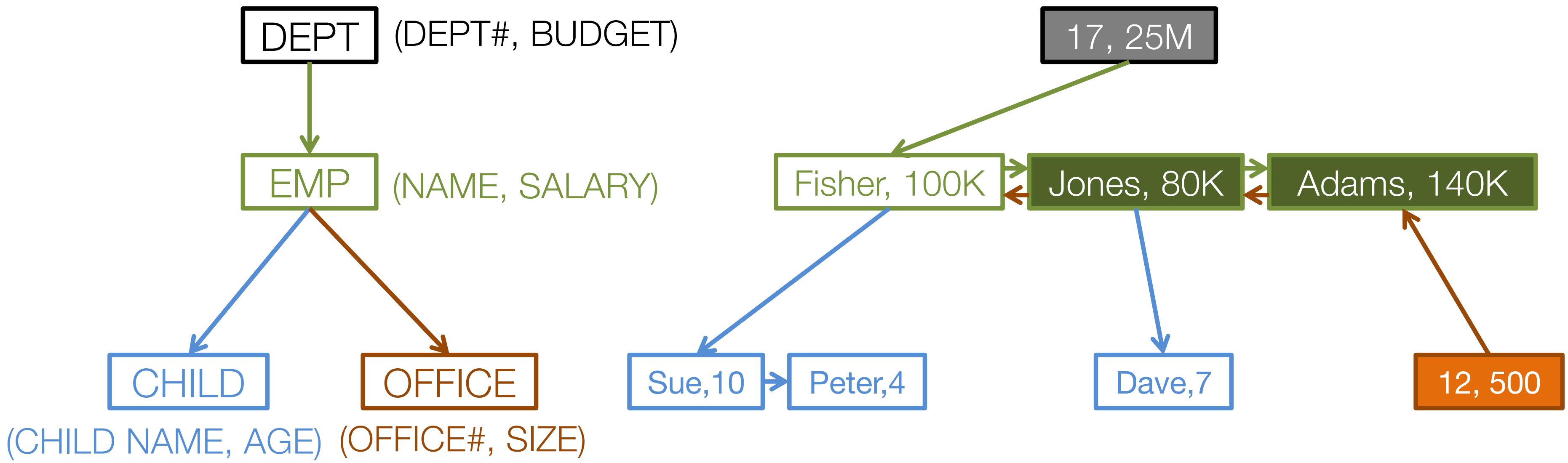
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

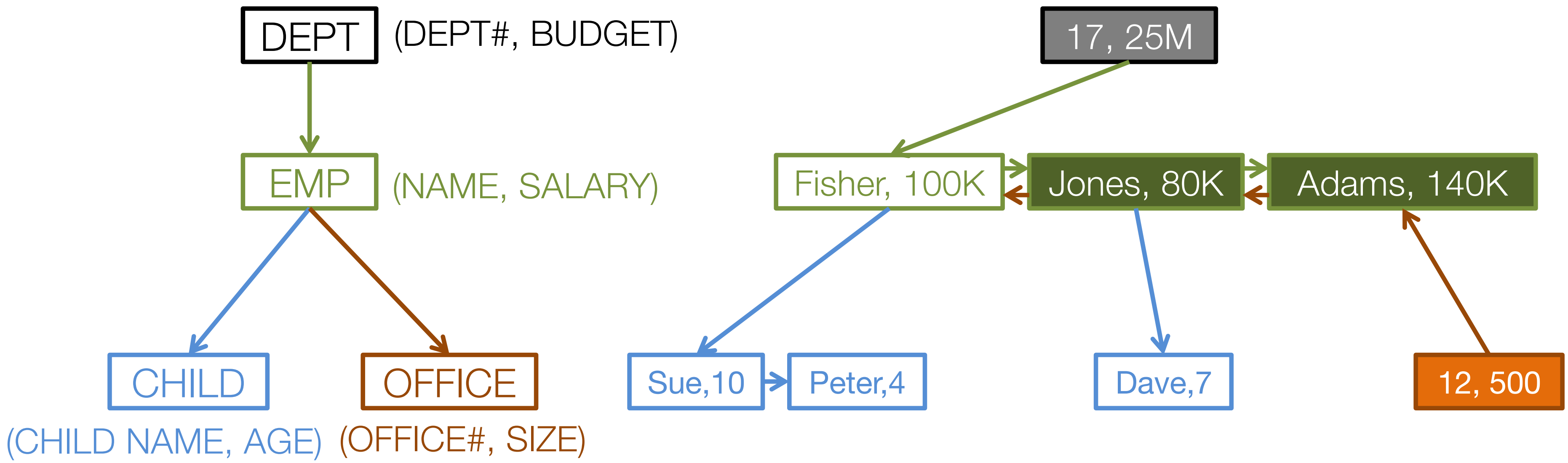
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

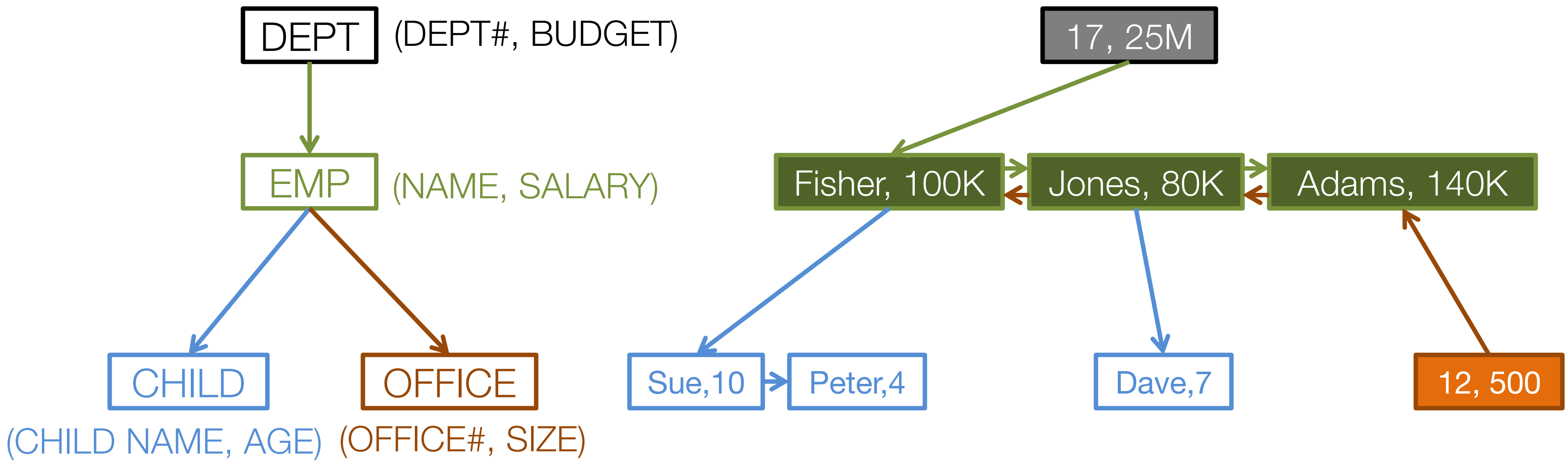
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output: 17

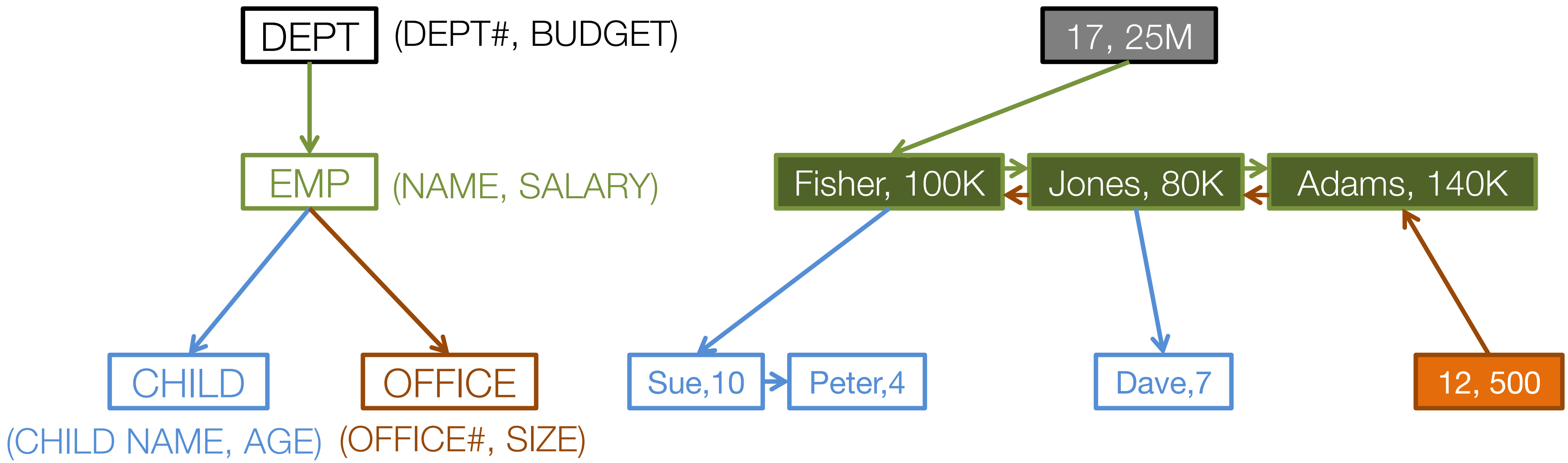
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output: 17

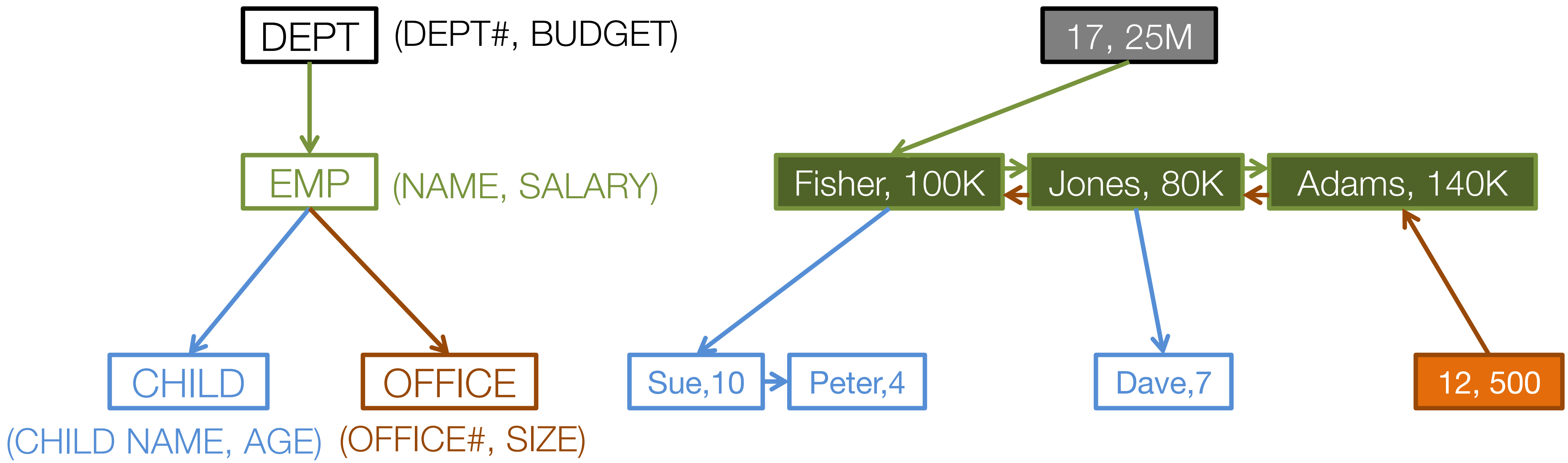
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

Output: 17

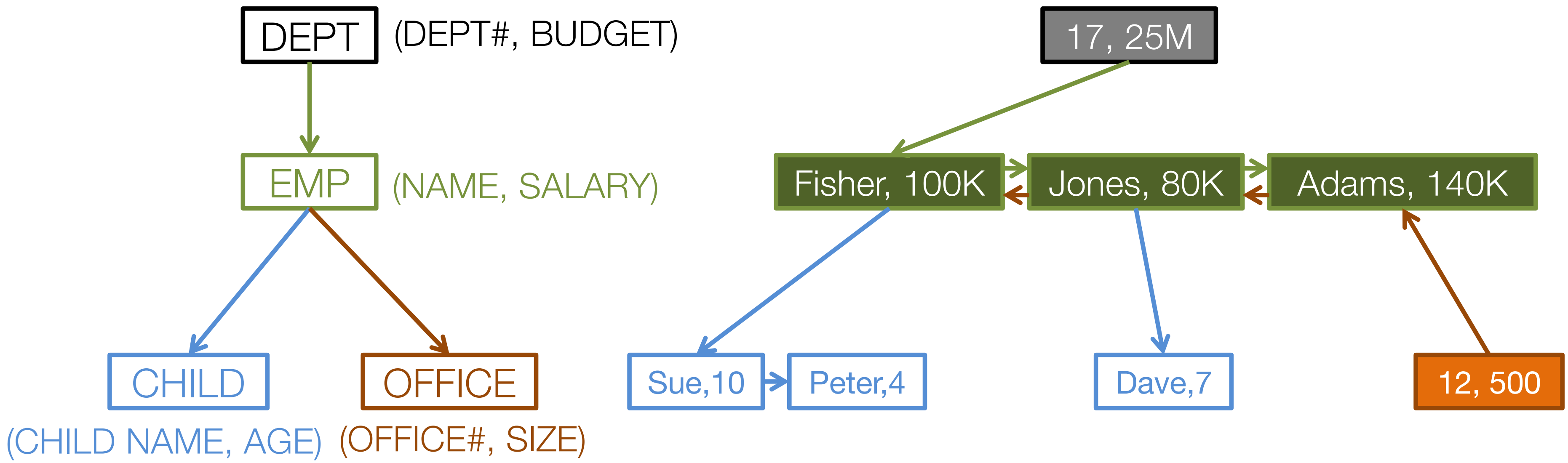
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

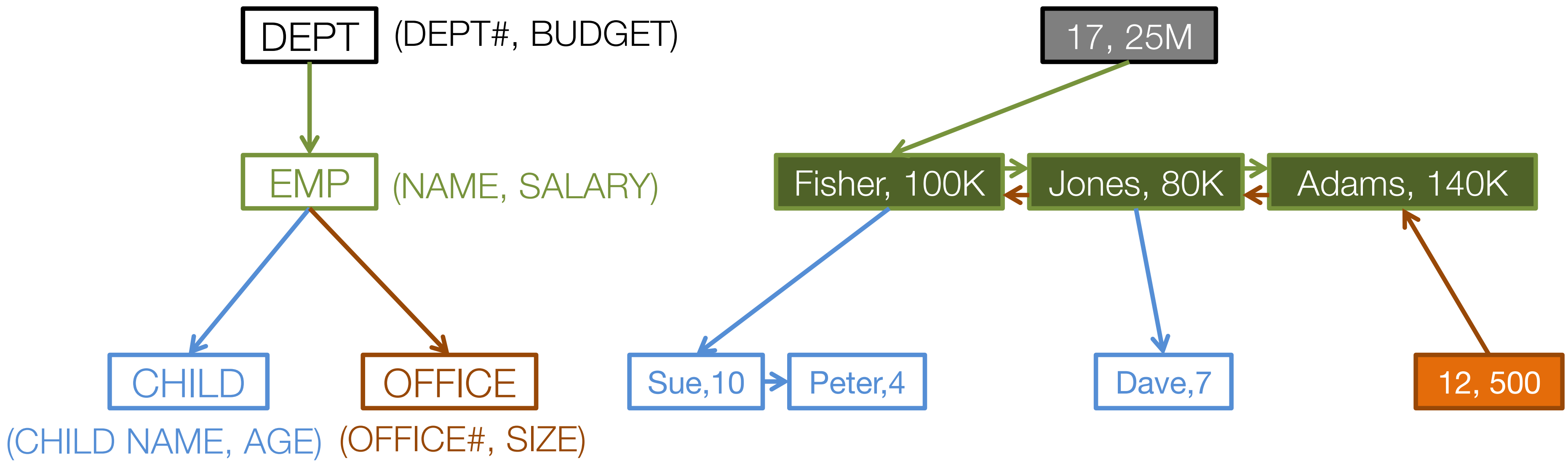
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return "no such office"
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return "done"
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return "employee exists which is not in a department"
          save department number
      GO TO LOOP
```

```
Output: 17
```

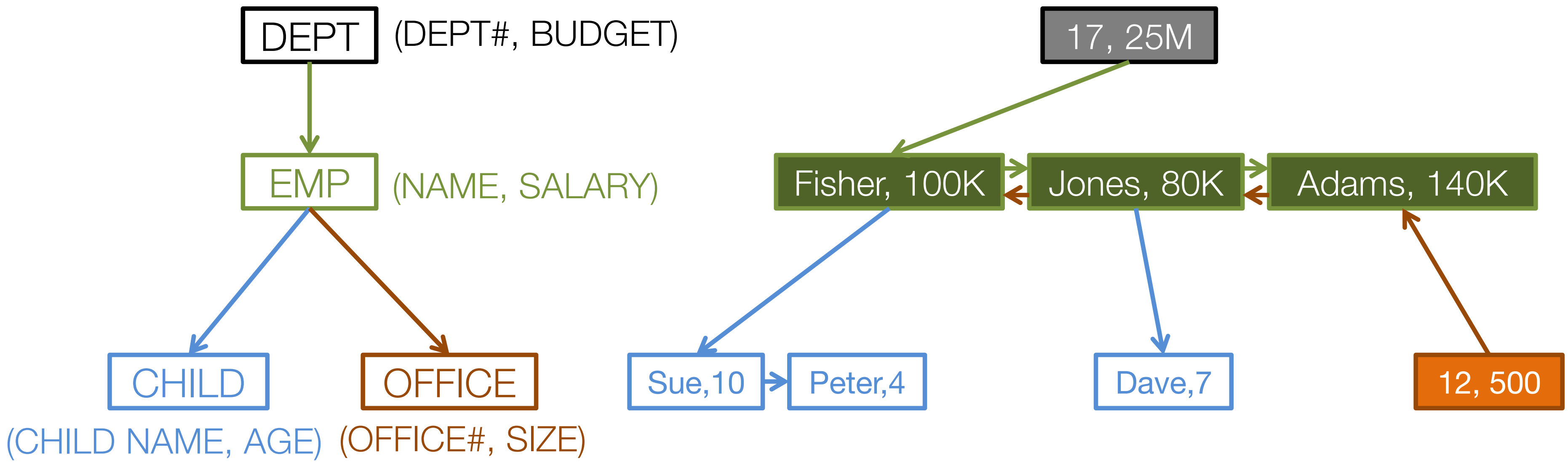
Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return “no such office”
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return “done”
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return “employee exists which is not in a department”
          save department number
      GO TO LOOP
```

```
Output: 17
```

Network Model




```
# find department numbers of all employees in office 12
      FIND OFFICE RECORD WITH OFFICE# = 12
          if failure; return “no such office”
LOOP  FIND NEXT MEMBER OF OCUPIED SET
          if failure; return “done”
      FIND OWNER OF CURRENT EMPLOYEE RECORD USING WORK SET
          if failure; return “employee exists which is not in a department”
          save department number
      GO TO LOOP
```

```
Output: 17
```

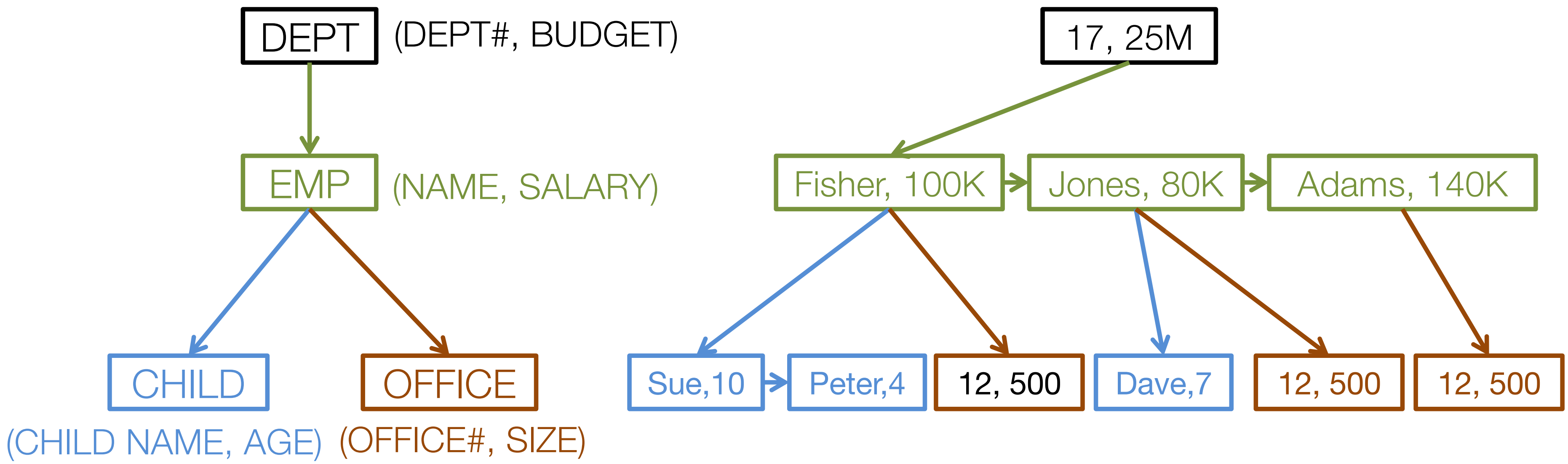
Data Dependence

Record-at-a-time Data Manipulation Language (DML)

Reflect physical data structures

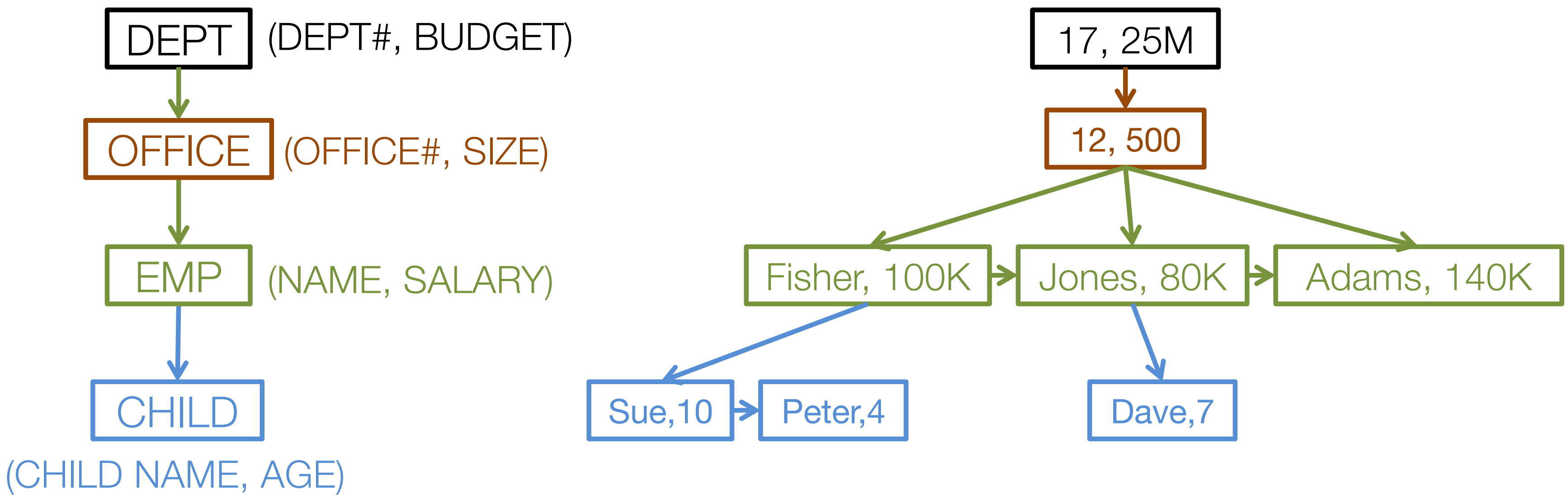
If you want to change the data organization you need to change query!

Example: Changing Data Representation



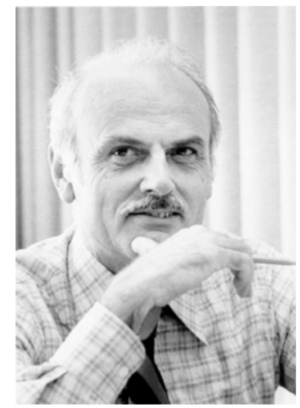

```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return “no such department”
      FIND 1st SON OF CURRENT RECORD
          if failure; return “no employee in this department”
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Example: Changing Data Representation




```
# find names of all employees in department 17
      FIND DEPT RECORD WHERE DEPT# = 17
          if failure; return "no such department"
      FIND 1st SON OF CURRENT RECORD
          if failure; return "no employee in this department"
LOOP      save name
      FIND NEXT BROTHER OF THE CURRENT RECORD WHICH IS OF SAME TYPE
      GO TO LOOP
```

Relational Database



1970 Edgar Codd's paper; probably the most influential paper in DB research

- » Set-at-a-time DML
- » Data independence: allows for schema and physical storage structures to change
  - *"as clear a paradigm shift as we can hope to find in computer science"* – Christos Papadimitriou
- » 1981 Turing Award

Relational Database:
Two key ideas

1. Store values only, no connections
   » Everything is a table
2. Declarative query language, leaves implementation and algorithm unspecified

Relational Model

Links represented as tables

DEPT	#	BUDGET
	17	25M

OFFICE	#	SIZE
	12	500

CHILD	C. NAME	C. NAME
	Sue	10
	Peter	4
	Dave	7

EMP	NAME	SALARY
	Fischer	100K
	Jones	80K
	Adams	140K

OFFSPRINGS	NAME	C. NAME
	Fischer	Sue
	Fischer	Peter
	Jones	Dave

WORKS	DEPT#	NAME
	17	Fisher
	17	Jones
	17	Adams

OCCUPIED	NAME	OFFICE#
	Fischer	12
	Jones	12
	Adams	12

```
# find department number of all employees in office 12
FIND ALL DEPT# in WORKS
    WHERE NAME = NAME IN OCCUPIED WHERE OFFICE# = 12
```

Data Independence

Separation into three levels:

» physical storage

» logical schema

» multiple views

Two levels of independence:

» physical data independence: you change the storage layout without affecting apps

» logical data independence: isolates apps from changes in logical schema (almost, as it can’t update views in general)

Data Independence

Critical for database evolution – allow databases live and evolve for a long time!

Need data independence when environment changes much faster than applications

- Environment: physical storage, machine speed, machine workload

First Relational Databases

Mid 70's: Codd's vision implemented by two projects: ancestors of essentially all today's commercial systems!

- » Ingres (UC Berkeley)
- » System R (IBM)

Lots of crosspollination between both groups

Ingres

1974-77, UC Berkeley: Stonebraker, Wong and many others

- 2015 Turing Award (Stonebraker)

Ancestor of:

- Ingres Corp (CA), CA-Universe, Britton-Lee, Sybase, MS SQL Server, Wang's PACE, Tandem Non-Stop SQL

System R

IBM San Jose (now Almaden)

- » 15 PhDs, including many Berkeley people:
  - Jim Gray (1st CS PhD @ Berkeley), Bruce Lindsay, Irv Traiger, Paul McJones, Mike Blasgen, Mario Schkolnick, Bob Selinger, Bob Yost
- » 1998 Turing Award (Gray)

Ancestor of:

- » IBM's SQL/DS & DB2, Oracle, HP's Allbase, Tandem Non-Stop SQL

Early 80's Commercialization

Ellison's Oracle beats IBM to market by reading white papers ;-)

IBM releases multiple RDBMSs, settles down to DB2

Gray (System R), Jerry Held (Ingres) and others join Tandem (Non-Stop SQL)

Kapali Eswaran starts EsVal, which led to HP Allbase and Cullinet

Relational Technology Inc (Ingres Corp), Britton-Lee/Sybase, Wang PACE grow out of Ingres group

CA releases CA-Universe, a commercialization of Ingres

Informix started by Cal alum Roger Sippl (no pedigree to research).

Teradata started by a Cal Tech alums, based on proprietary networking technology

Discussion

R System Goals

1. To provide a **high-level, non-navigational user interface** for maximum user productivity and data independence.
2. To support **different** types of database **use** including programmed **transactions**, **ad hoc queries**, and **report** generation.
3. To support a **rapidly changing database environment**, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the database without stopping the system.
4. To support a population of **many concurrent users**, with mechanisms to protect the integrity of the database in a concurrent-update environment.
5. To provide a means of **recovering** the contents of the database to a consistent state after a failure of hardware or software.
6. To provide a flexible mechanism whereby **different views** of stored data can be defined and various users can be authorized to query and update these views.
7. To support all of the above functions with a level of **performance comparable to existing lower-function database systems.**

R System Goals

1. Easy of use
2. Unifying abstraction
3. Evolvability (of database)
4. Concurrency
5. Fault tolerance
6. Access control (and flexibility)
7. Performance

1. To provide a **high-level, non-navigational user interface** for maximum user productivity and data independence.
2. To support **different** types of database **use** including programmed **transactions**, **ad hoc queries**, and **report** generation.
3. To support a **rapidly changing database environment**, in which tables, indexes, views, transactions, and other objects could easily be added to and removed from the database without stopping the system.
4. To support a population of **many concurrent users**, with mechanisms to protect the integrity of the database in a concurrent-update environment.
5. To provide a means of **recovering** the contents of the database to a consistent state after a failure of hardware or software.
6. To provide a flexible mechanism whereby **different views** of stored data can be defined and various users can be authorized to query and update these views.
7. To support all of the above functions with a level of **performance comparable to existing lower-function database systems.**

Development

Expect to throw out the 1st version of the system

Why?

Development

Expect to throw out the 1st version of the system

Authors very familiar with

- » What they want to build
- » Implementation challenges

Similar to Unix:

- » Ken Thomson and Dennis Ritchie both worked on Multics

Query Optimization:
Phase Zero vs One

Phase Zero focus: optimize complex queries

Phase One focus: optimize simple, most common queries

Query Optimization: Phase Zero vs One

Phase Zero: “predicate locks” deemed to complicated

Phase One: per object locks, albeit hierarchical and multiple granularity

Storage: Phase Zero vs One

Phase Zero: single user, XRM

- » Values of each column stored in a separate domain
- » Each field contains TID of corresponding domain/value
- » Inversions: mapping between values and TIDs

Phase One: multiuser, RSS

- » Tuple contains values
- » Indexes on one, more, or combination of columns

Cost Based Optimizer: Phase Zero vs One

Phase Zero: per-tuple cost

Phase One: combination of

- » # of I/Os
- » # of calls (CPU activity)

Questions

- » How well does it work?
- » Do you expect CPU to still be bottleneck today?
- » Caveats?

Others

Interpretation vs. compilation?

Three levels of transactions?

Shadow pages vs write ahead logging?

Unix vs. System R

UNIX paper: "The most important job of UNIX is to provide a file system”

» UNIX and System R are both "information management" systems!

Both also provide programming APIs for apps

Unix vs. System R: Goals

Ease of use:

- Unix: "most important characteristics of the system are its **simplicity**, **elegance**, and **ease** of use."
- System R: "The system was mostly used in applications for which **ease** of installation, a **high-level user language**, and an ability to rapidly reconfigure the database were important requirements."

Unix vs. System R: Reuse

Unix

- » Written in the developer language, C
- » Directories as files
- » I/O naming as file naming
- » Same security mechanism for I/O devices & files
- » Shell, just another process

System R?

Unix vs. System R: Reuse

Unix

- Written in the developer language, C
- Directories as files
- I/O naming as file naming
- Same security mechanism for I/O devices & files
- Shell, just another process

System R?

- Relations as tables
- Catalogs as tables
- Views: any SQL query to be used as view definition
- …

Unix vs. System R: Philosophy

Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)

UNIX main function: *present hardware to computer programmers*

- small *elegant* set of mechanisms, and abstractions for developers (i.e. C programmers)

System R: *manage data for application programmer*

- complete system that insulated programmers (i.e. SQL + scripting) from the system, while guaranteeing clearly defined *semantics* of data and queries.

Unix vs. System R: Philosophy

Bottom-Up (elegance of system) vs. Top-Down (elegance of semantics)

Affects where the complexity goes: to system, or end-programmer?

Which one is better? In what environments?

Summary

Advantages of relational databases:

» Can change data layout without breaking things
» Don't need to worry about coming up with algo.
» Don't need to be a genius to query.
» More readable queries (no convoluted algo query).
» Database can come up with algo; algo and layout can be dynamic over time
» Apps don't need to be recompiled. Just upgrade database and get speedup.
» Database can adapt to new hardware, apps don't need to worry

Summary

Disadvantages of relational databases:

» Hard to express certain things in declarative form

» Could be slower for small simple imperative queries

Backup

Different Challenges

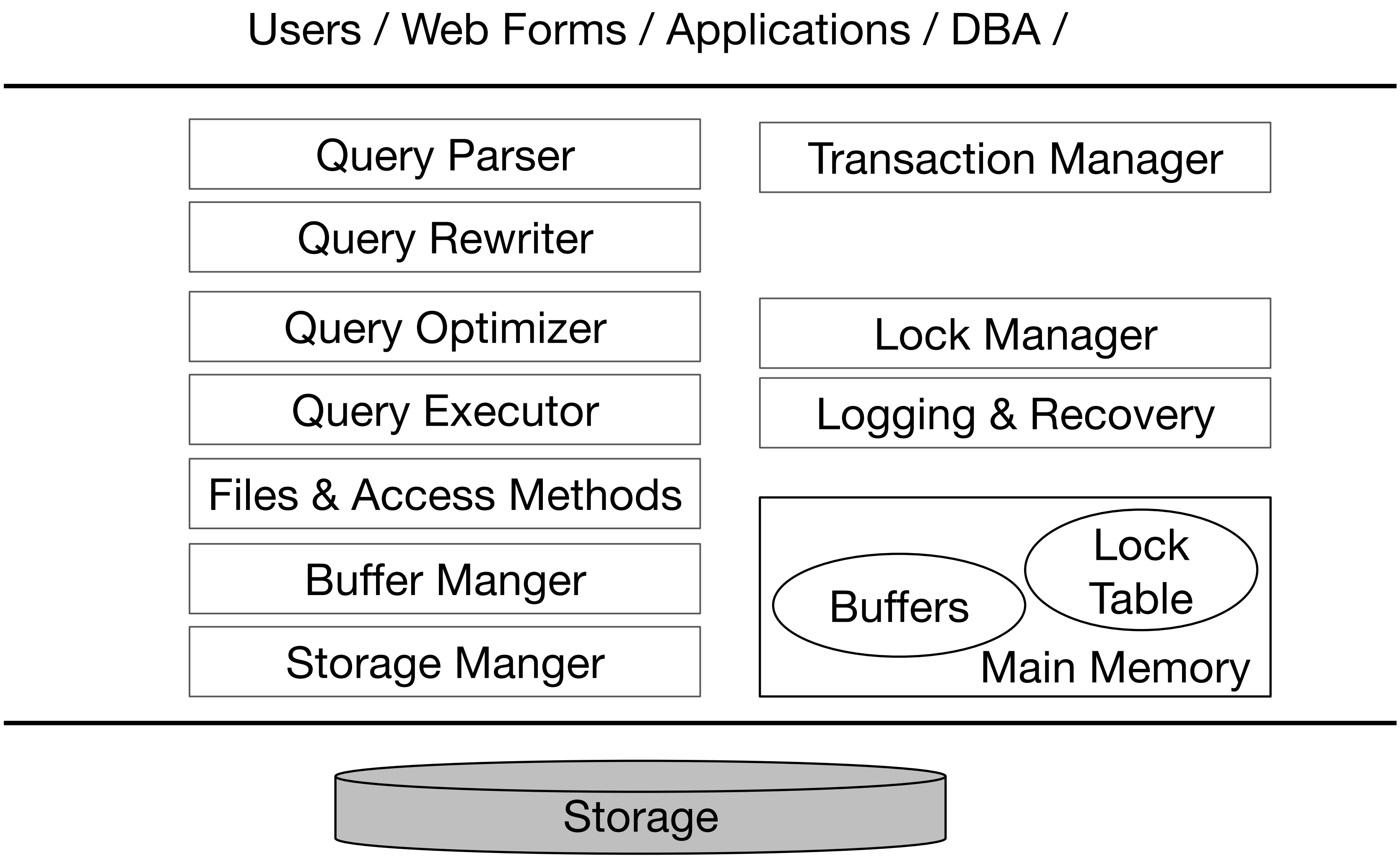
Achilles' heel of RDBMSs: closed box

- » Cannot leverage technology without going through the full SQL stack
- » One solution: make the system extensible, convince the world to download code into the DBMS
- » Another solution: componentize the system (hard, RSS is hard to bust up, due to transaction semantics)

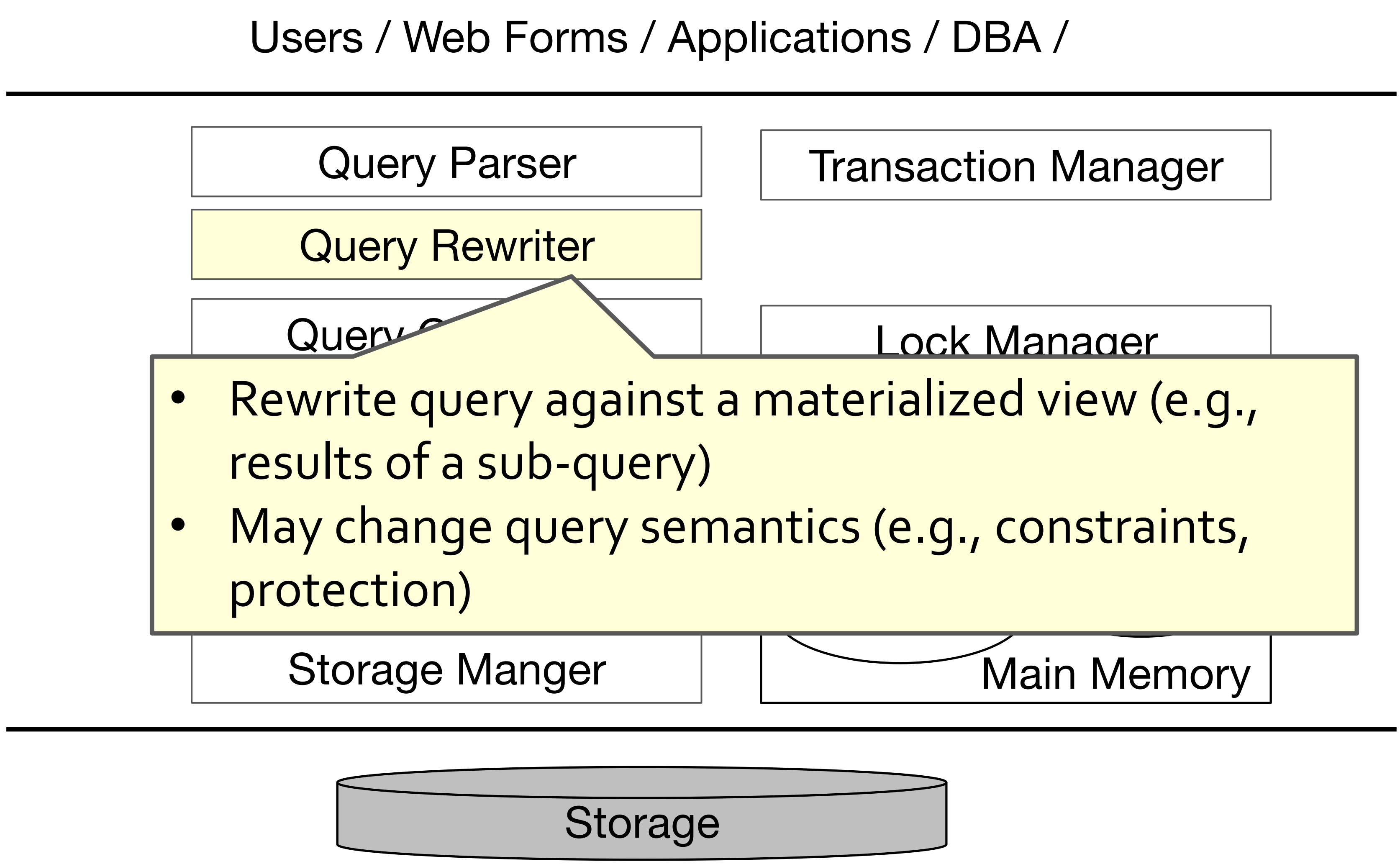
Achilles' heel of OSes: hard to get "right" level of abstraction

- » Many UNIX abstractions (e.g. virtual memory) too high level, hide too much detail
  - In contrast, too low a level can cause too much programmer burden
- » One solution: make the system extensible, convince fancy apps to download code into the OS
- » Another solution: componentize the system (hard, due to protection)
  - But lot’s of work on this, e.g., Microkernel

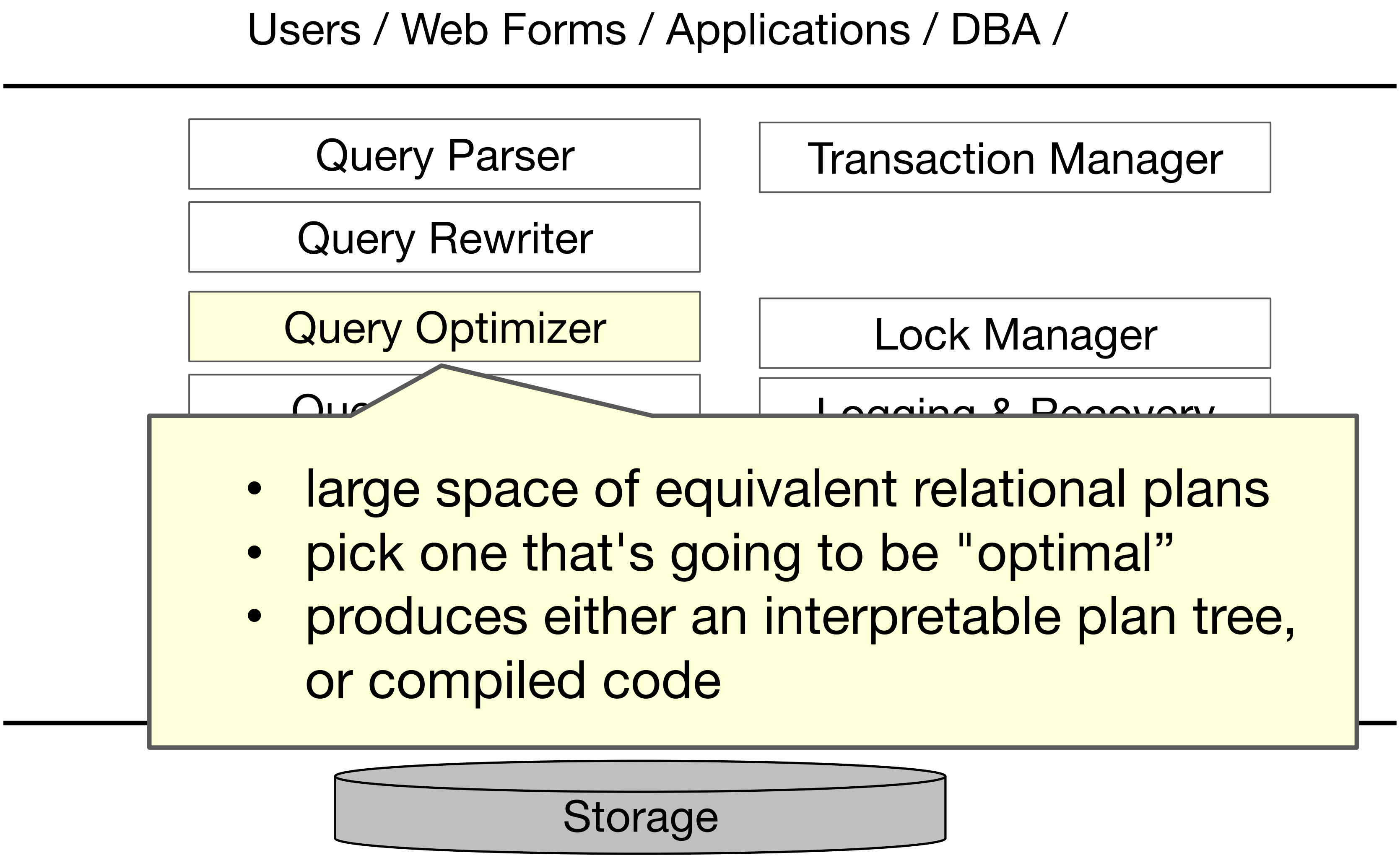
Database Architecture



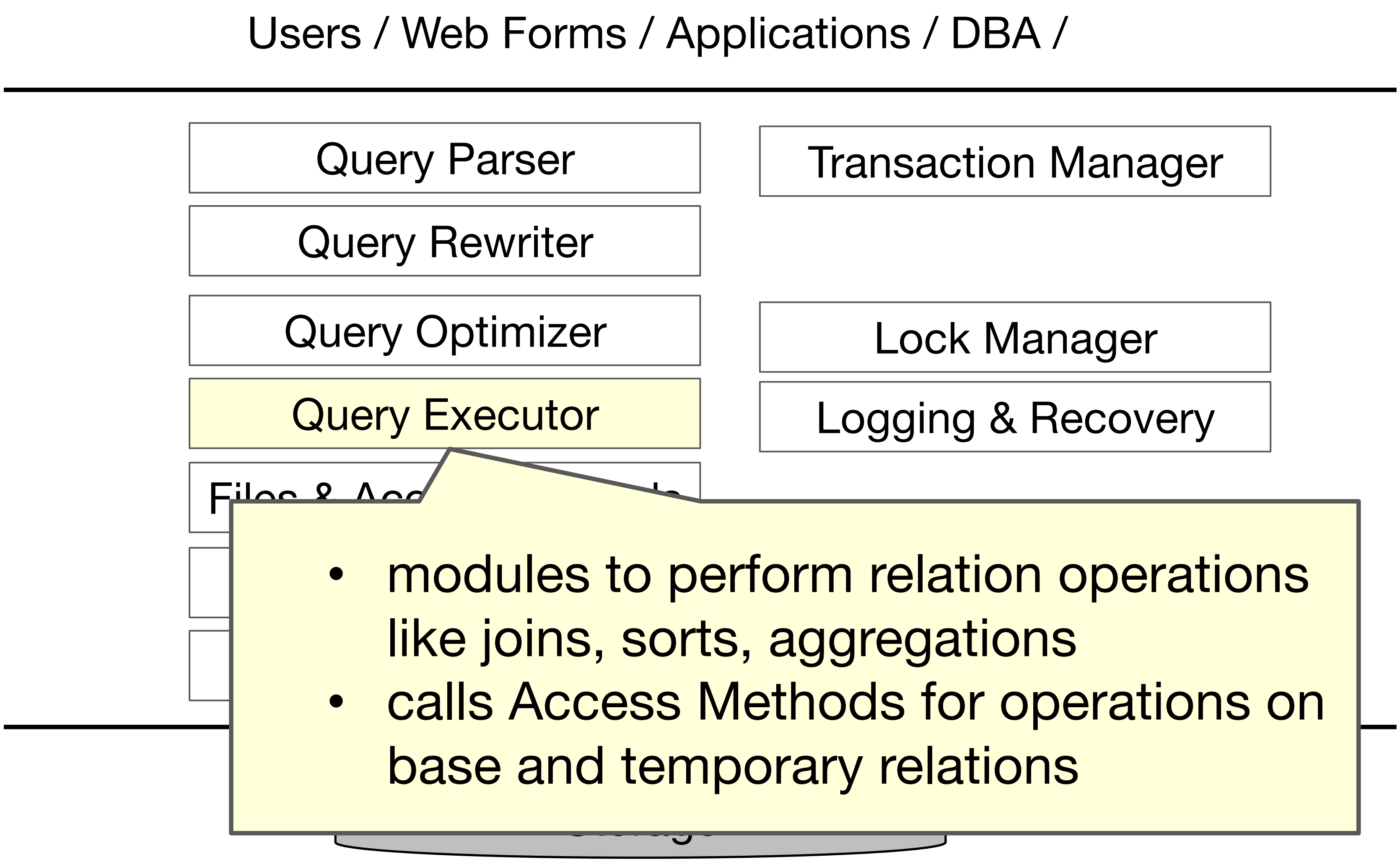
Database Architecture



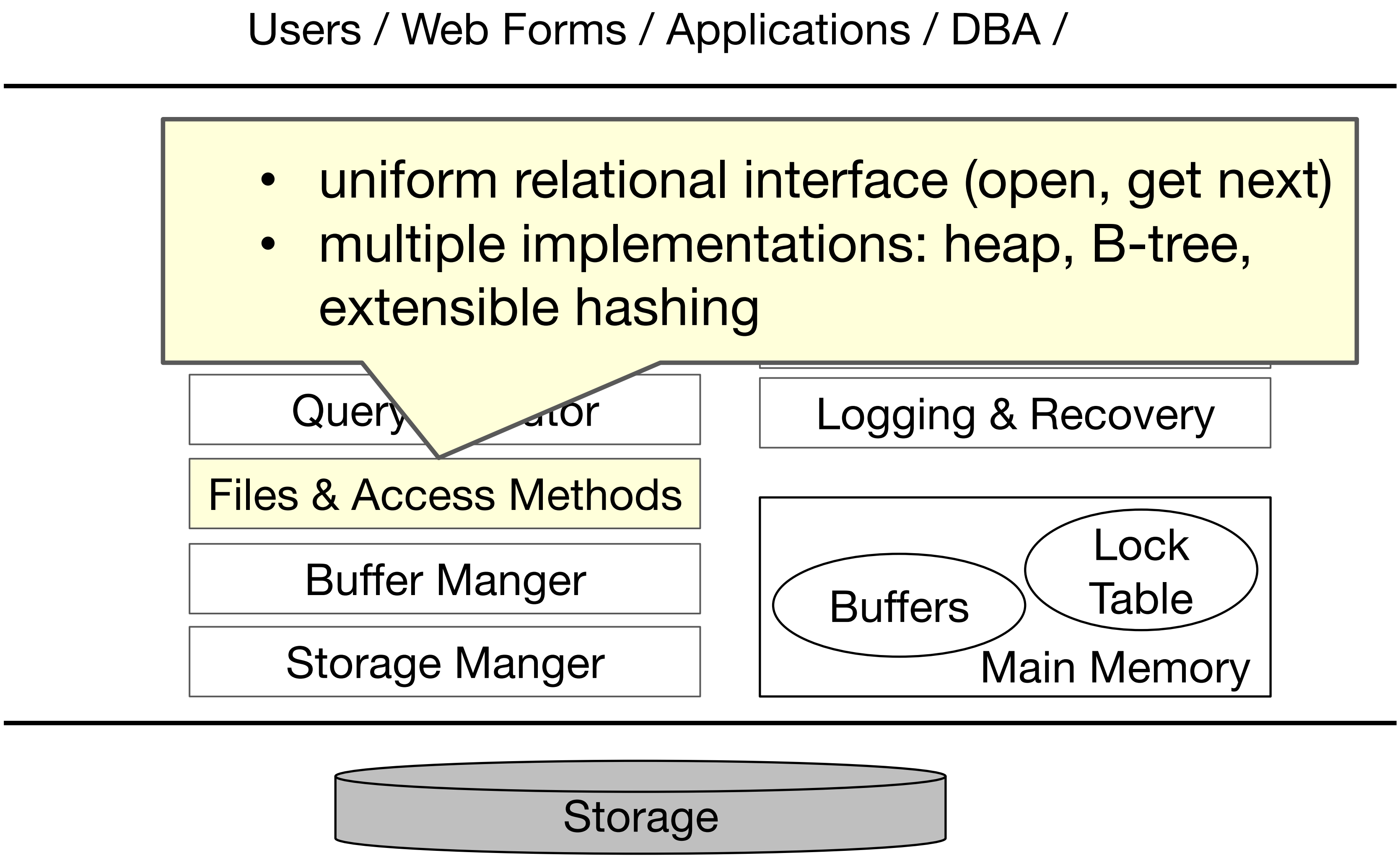
Database Architecture



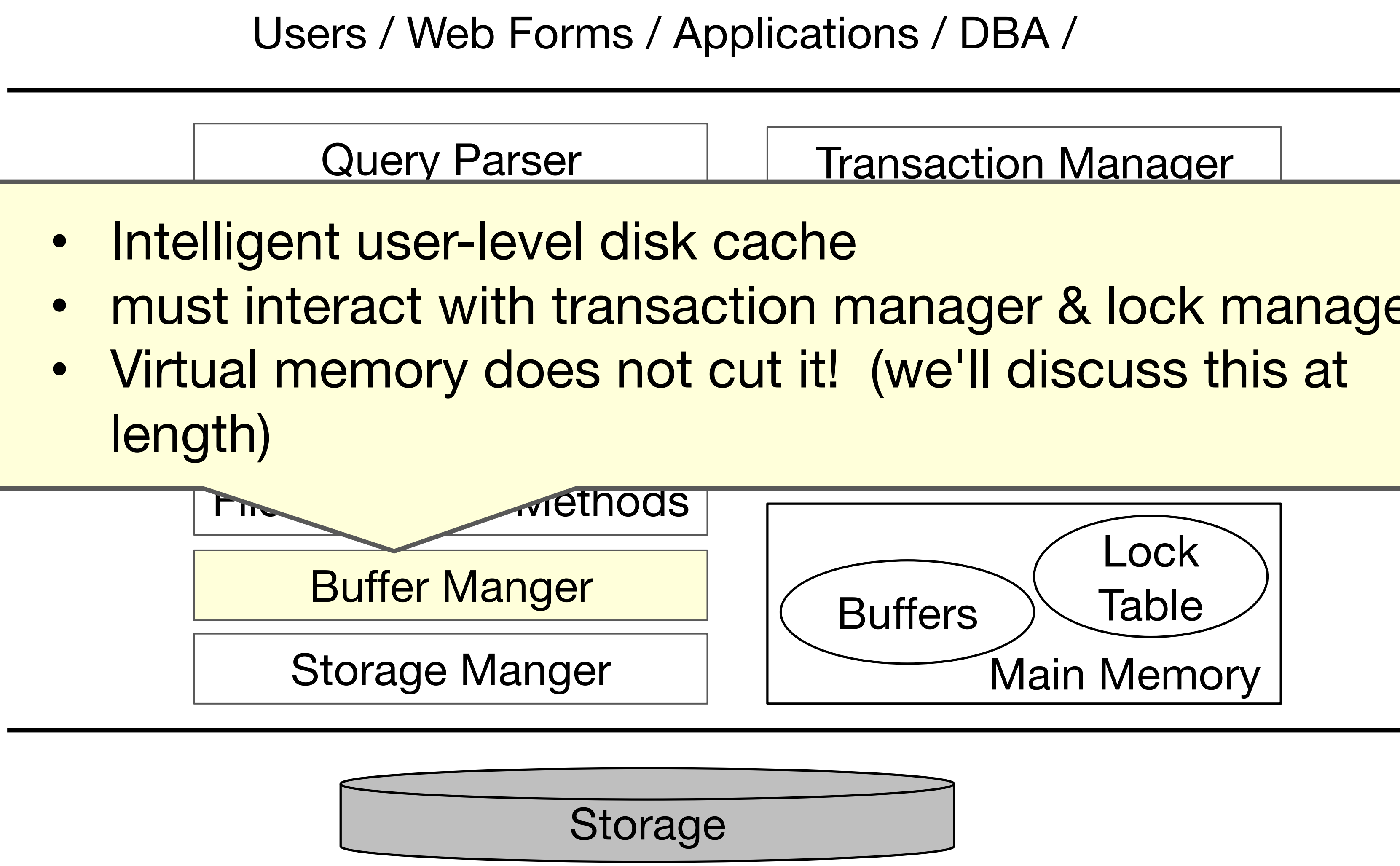
Database Architecture



Database Architecture

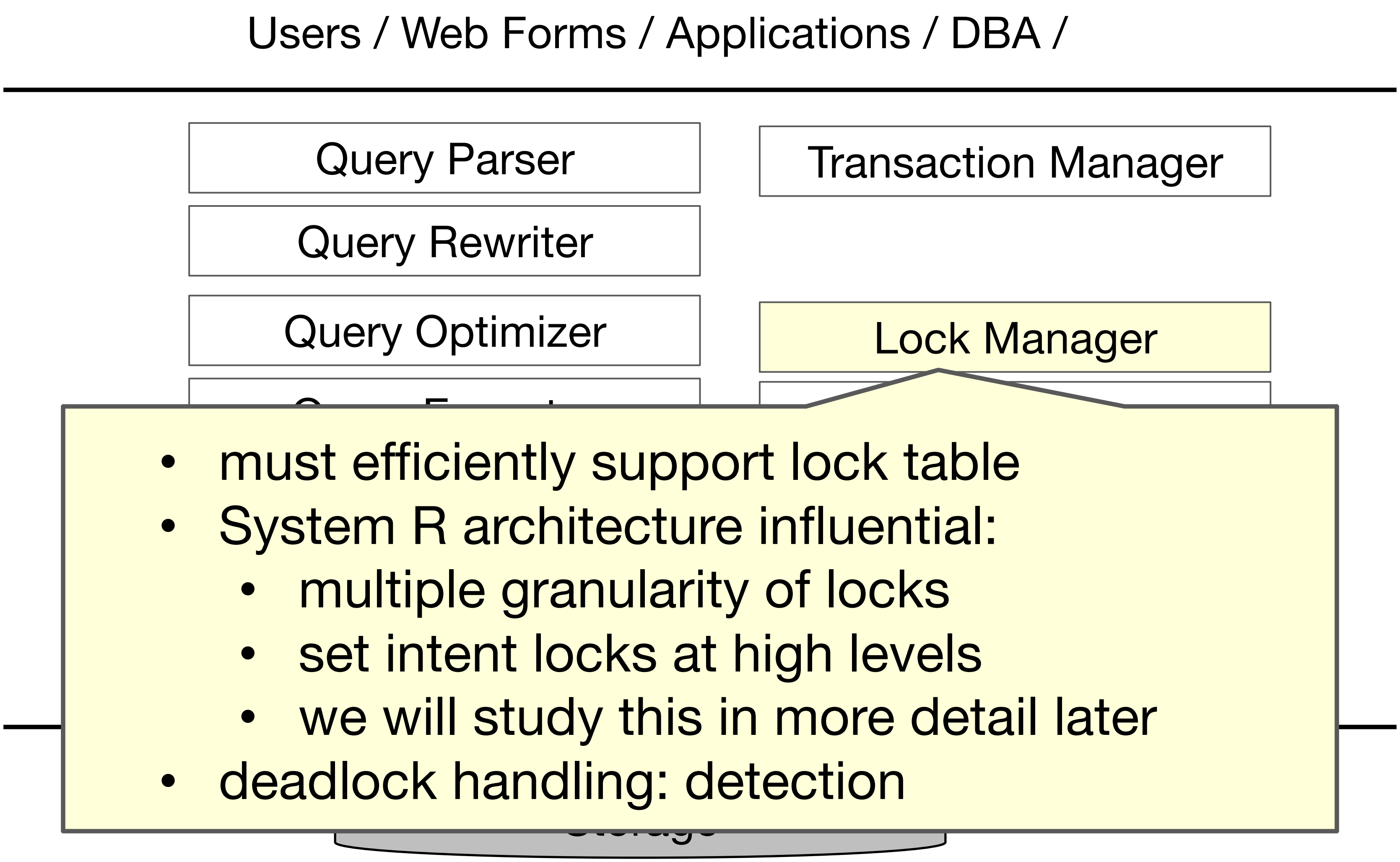


Database Architecture



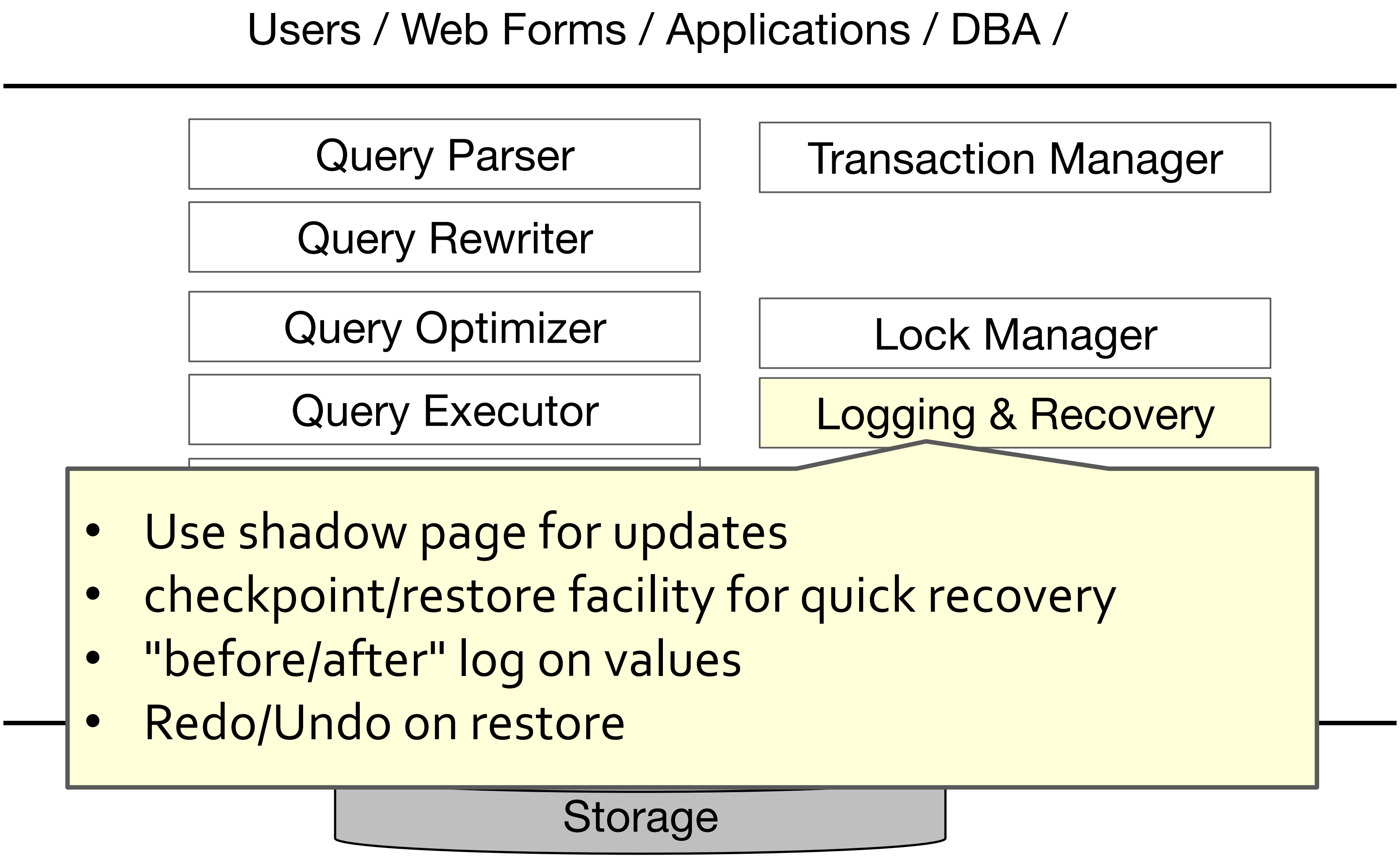
- Intelligent user-level disk cache
- must interact with transaction manager & lock manager
- Virtual memory does not cut it! (we'll discuss this at length)

Database Architecture



- must efficiently support lock table
- System R architecture influential:
  - multiple granularity of locks
  - set intent locks at high levels
  - we will study this in more detail later
- deadlock handling: detection

Database Architecture



Discussion: Shadow Pages

Shadow pages:

- » New version is created for each page that is updated
- » Periodically new page is checkpointed on disk
- » “before/after” logs recording all database changes
- » On failure, revert to “old” page and use log to redo committed transactions and undo incomplete ones

Write Ahead Log (WAL):

- » Keep a log of all database updates
- » Write each update before writing back to disk the updates

Tradeoffs?

Others: Transactions

Level 1: Transactions can read uncommitted transactions but not write

Level 2: Transactions acquire lock for each reads but releases it right after reading

- » Another transaction may update a value between two reads

Level 3: Once a transaction acquires a read lock it keeps it until the end

Discussion?

System R Paper Nuggets

Interpretation vs. compilation

R Systems use compilation: compiler assembles from about 100 code fragments specially tailored for processing a given SQL query”

System R Paper Nuggets

Component failure as common case

Three failure cases:

1. System failure
2. Media (disk) failure
3. Transaction failure