Database Implementation For Modern Hardware

Thomas Neumann

Introduction

Database Management Systems (DBMS) are extremely important

- used in nearly all commercial data management
- very large data sets
- valuable data

Key challenges:

- scalability to huge data sets
- reliability
- concurrency

Results in very complex software.

About This Lecture

Goals of this lecture

- learning how to build a modern DBMS
- understanding the internals of existing DBMSs
- understanding the effects of hardware behavior

Rough structure of the lecture

- 1. the classical DBMS architecture
- 2. efficient query processing
- 3. adapting the architecture to hardware trends

Literature

- Theo Härder, Erhard Rahm: *Datenbanksysteme Konzepte und Techniken der Implementierung*. Springer-Verlag, 2001.
- Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom: Database Systems: The Complete Book. Prentice-Hall, 2008.
- Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993

Unfortunately mainly cover the classical architecture.

Motivational Example

Why is a DBMS different from most other programs?

- many difficult requirements (reliability etc.)
- but a key challenge is scalability

Motivational example

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Looks simple...

Motivational Example (2)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Simple if both fit in main memory

Motivational Example (2)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Simple if both fit in main memory

- sort both lists and intersect
- or put one in a hash table and probe
- or build index structures
- or ...

Note: pairwise comparison is not an option! $O(n^2)$

Motivational Example (3)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Slightly more complex if only one fits in main memory

Motivational Example (3)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Slightly more complex if only one fits in main memory

- load the smaller list into memory
- build index structure/sort/hash/...
- scan the larger list
- search for matches in main memory

Code still similar to the pure main-memory case.

Motivational Example (4)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Difficult if neither list fits into main memory

Motivational Example (4)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Difficult if neither list fits into main memory

- no direct interaction possible
- sorting works, but already a difficult problem
- or use some kind of partitioning scheme
- breaks the problem into smaller problem
- until main memory size is reached

Code significantly more involved.

Motivational Example (5)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Hard if we make no assumptions about L_1 and L_2 .

Motivational Example (5)

Given two lists L_1 and L_2 , find all entries that occur on both lists.

Hard if we make no assumptions about L_1 and L_2 .

- tons of corner cases
- a list can contain duplicates
- a single duplicate might exceed main memory!
- breaks "simple" external memory logic
- multiple ways to solve this
- but all of them somewhat involved
- and a DBMS must not make assumptions about its data!

Code complexity is very high.

Motivational Example (6)

Designing scalable algorithm is a hard problem

- must cope with very large instances
- hard even in main memory
- billions of data items
- rules out any $O(n^2)$ algorithm
- external algorithms are even harder

This requires a careful software architecture.

Set-Oriented Processing

Small applications often loop over their data

- one for loop accesses all item x,
- for each item, another loop access item y,
- then both items are combined.

This kind of code of code feels "natural", but is bad

- $\Omega(n^2)$ runtime
- does not scale

Instead: set oriented processing. Perform operations for large batches of data.

Relational Algebra

Notation:

- A(e) attributes of the tuples produces by e
- $\mathcal{F}(e)$ free variables of the expression e
- binary operators $e_1 heta e_2$ usually require $\mathcal{A}(e_1) = \mathcal{A}(e_2)$

$$\begin{array}{ll} e_1 \cup e_2 & \text{union, } \{x | x \in e_1 \lor x \in e_2\} \\ e_1 \cap e_2 & \text{intersection, } \{x | x \in e_1 \land x \in e_2\} \\ e_1 \setminus e_2 & \text{difference, } \{x | x \in e_1 \land x \notin e_2\} \\ \rho_{a \to b}(e) & \text{rename, } \{x \circ (b : x.a) \setminus (a : x.a) | x \in e\} \\ \Pi_A(e) & \text{projection, } \{\circ_{a \in A}(a : x.a) | x \in e\} \\ e_1 \times e_2 & \text{product, } \{x \circ y | x \in e_1 \land y \in e_2\} \\ \sigma_p(e) & \text{selection, } \{x | x \in e \land p(x)\} \\ e_1 \bowtie_p e_2 & \text{join, } \{x \circ y | x \in e_1 \land y \in e_2 \land p(x \circ y)\} \end{array}$$

per definition set oriented. Similar operators also used bag oriented (no implicit duplicate removal).

Relational Algebra - Derived Operators

Additional (derived) operators are often useful:

 $\begin{array}{ll} e_{1} \bowtie_{e_{2}} & \text{natural join, } \{x \circ y_{|\mathcal{A}(e_{2}) \setminus \mathcal{A}(e_{1})} | x \in e_{1} \land y \in e_{2} \land x =_{|\mathcal{A}(e_{1}) \cap \mathcal{A}(e_{2})} y\} \\ e_{1} \div e_{2} & \text{division, } \{x_{|\mathcal{A}(e_{1}) \setminus \mathcal{A}(e_{2})} | x \in e_{1} \land \forall y \in e_{2} \exists z \in e_{1} : \\ & y =_{|\mathcal{A}(e_{2})} z \land x =_{|\mathcal{A}(e_{1}) \setminus \mathcal{A}(e_{2})} z\} \\ e_{1} \bowtie_{p} e_{2} & \text{semi-join, } \{x | x \in e_{1} \land \exists y \in e_{2} : p(x \circ y)\} \\ e_{1} \bowtie_{p} e_{2} & \text{outer-join, } \{x | x \in e_{1} \land \exists y \in e_{2} : p(x \circ y)\} \\ e_{1} \bowtie_{p} e_{2} & \text{outer-join, } (e_{1} \bowtie_{p} e_{2}) \cup \{x \circ \circ_{a \in \mathcal{A}(e_{2})}(a : null) | x \in (e_{1} \bowtie_{p} e_{2})\} \\ e_{1} \bowtie_{p} e_{2} & \text{full outer-join, } (e_{1} \bowtie_{p} e_{2}) \cup (e_{2} \bowtie_{p} e_{1}) \end{array}$

Relational Algebra - Extensions

The algebra needs some extensions for real queries:

- map/function evaluation $\chi_{a:f}(e) = \{x \circ (a : f(x)) | x \in e\}$
- group by/aggregation

$$\Gamma_{A;a:f}(e) = \{x \circ (a:f(y)) | x \in \Pi_A(e) \land y = \{z | z \in e \land \forall a \in A : x.a = z.a\}\}$$

• dependent join (djoin). Requires $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$ $e_1 \bowtie_p e_2 = \{x \circ y | x \in e_1 \land y \in e_2(x) \land p(x \circ y)\}$

Set-Oriented Processing (2)

Processing whole batches of tuples is more efficient:

- can prepare index structures
- or re-organize the data
- sorting/hashing
- runtime ideally O(nlogn)

Many different algorithms, we will look at them later.

Traditional Assumptions

Historically, DBMS are designed for the following scenario:

- data is much larger than main memory
- I/O costs dominate everything
- random I/O is very expensive

This led to a very conservative, but also very scalable design.

Hardware Trends

Hardware development changed some of the assumptions

- main memory size is increasing
- servers with 1TB main memory are affordable
- flash storage reduces random I/O costs

• ...

This has consequences for DBMS design

- CPU costs become more important
- often I/O is eliminated or greatly reduced
- the old architecture becomes suboptimal

But this is more evolution than revolution. Many of the old techniques are still required for scalability reasons.

Goals

Ideally, a DBMS

- handles arbitrarily large data sets efficiently
- never loses data
- offers a high-level API to manipulate and retrieve data
- shields the application from the complexity of data management
- offers excellent performance for all kinds of queries and all kinds of data

This is a very ambitious goal!

In many cases indeed reached, but implies complexity.

Overview

1. The Classical Architecture

- 1.1 storage
- 1.2 access paths
- 1.3 transactions and recovery
- 2. Efficient Query Processing
 - 2.1 set oriented query processing
 - 2.2 algebraic operators
 - 2.3 code generation
- 3. Designing a DBMS for Modern Hardware
 - 3.1 re-designing storage
 - 3.2 optimizing cache locality
 - 3.3 main memory databases