Data Management for Data Science

Lecture 7: Wrapping up RDBMS

Prof.Asoc.Endri Raço

Today’s Lecture

1. Finish SQL

2. Overview of an RDBMS

3. Transactions and ACID

1. SQL (Aggregation and Group By)

Aggregation

```
SELECT AVG(price)
FROM   Product
WHERE  maker = “Toyota”
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

Note: Same as COUNT(*). Why?

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

More Examples

Purchase(product, date, price, quantity)

```
SELECT SUM(price * quantity)
FROM   Purchase
```

What do these mean?

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = ‘bagel’
```

⇒ 50 (= 1*20 + 1.50*20)

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let's see what this means…

Grouping and Aggregation

Semantics of the query:

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM →

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

<table>
<tr><th>Product</th><th>Date</th><th>Price</th><th>Quantity</th></tr>
<tr><td rowspan="2">Bagel</td><td>10/21</td><td>1</td><td>20</td></tr>
<tr><td>10/25</td><td>1.50</td><td>20</td></tr>
<tr><td rowspan="2">Banana</td><td>10/3</td><td>0.5</td><td>10</td></tr>
<tr><td>10/10</td><td>1</td><td>10</td></tr>
</table>

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT →

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

General form of Grouping and Aggregation

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

Why?

- S = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1,\ldots,R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

```
SELECT    S
FROM      R_1,…,R_n
WHERE     C_1
GROUP BY  a_1,…,a_k
HAVING    C_2
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Group-by v.s. Nested Query

Author(login, name)

Wrote(login, url)

- Find authors who wrote $\geq$ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE  COUNT(
          SELECT Wrote.url
    FROM   Wrote
    WHERE  Author.login = Wrote.login) > 10
```

This is
SQL by
a novice

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT   Author.name
FROM     Author, Wrote
WHERE    Author.login = Wrote.login
GROUP BY Author.name
HAVING   COUNT(Wrote.url) > 10
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested:* How many times do we do a SFW query over all of the Wrote relations?

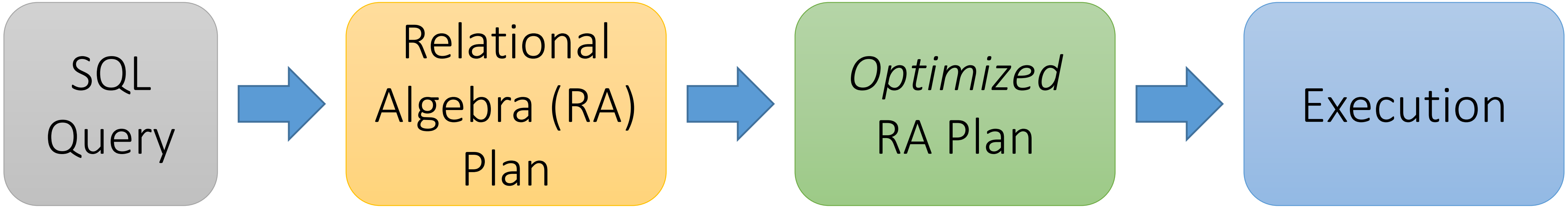
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be <u>much</u> more efficient!

2. Overview of an RDBMS

RDBMS Architecture

How does a SQL engine work ?



Declarative query (from user)

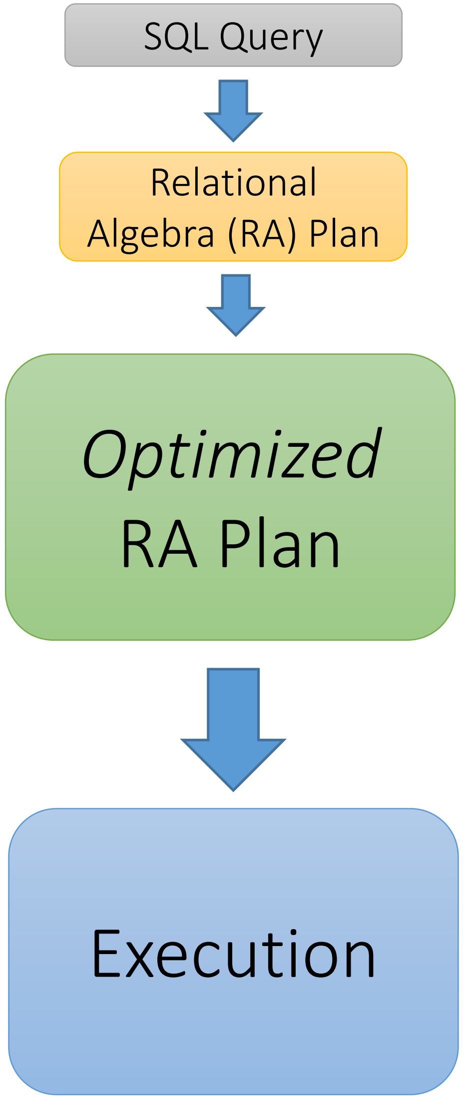
Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

Logical vs. Physical Optimization

- **Logical optimization (we will only see this one):**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*
- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



Recall: Logical Equivalence of RA Plans

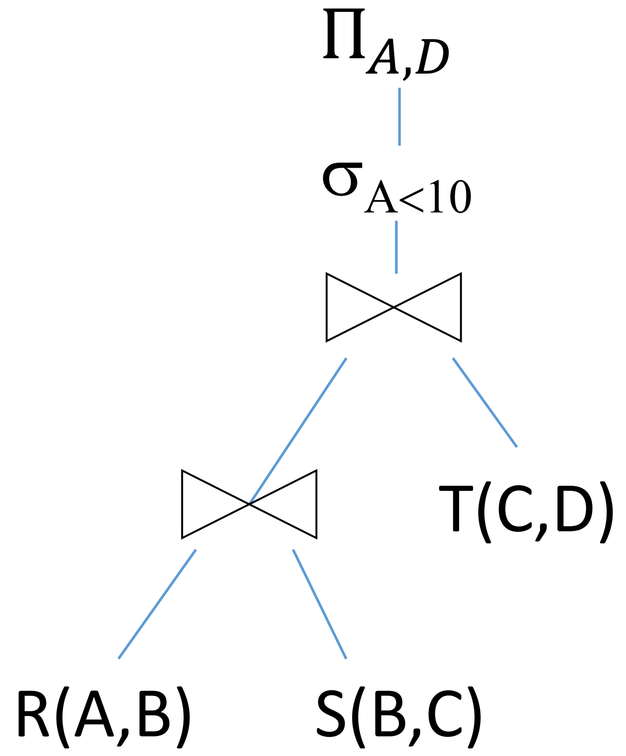
- Given relations R(A,B) and S(B,C):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \,?\!= \Pi_B(\sigma_{A=5}(R))$

Translating to RA

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections**” and “pushing down **projections.”**

- **Intuition:** We will have fewer tuples in a plan.
  - Could fail if the selection condition is very expensive (say runs some image processing algorithm).
  - Projection could be a waste of effort, but more rarely.

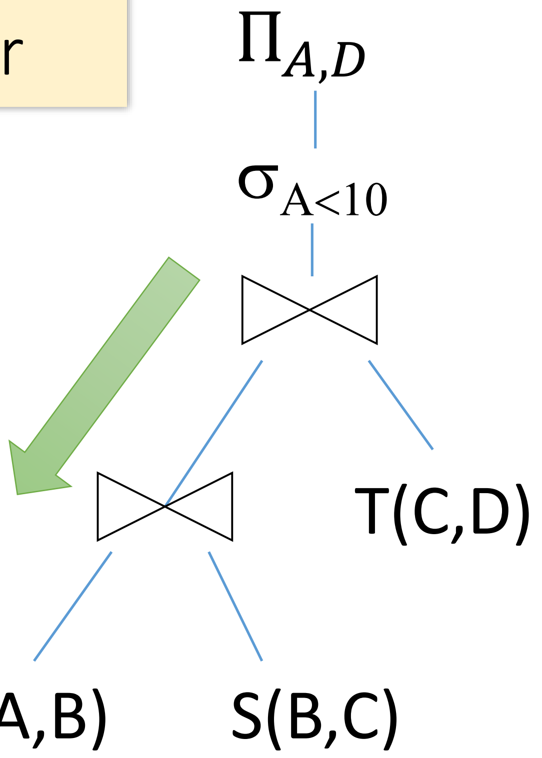
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$

Push down selection on A so it occurs earlier



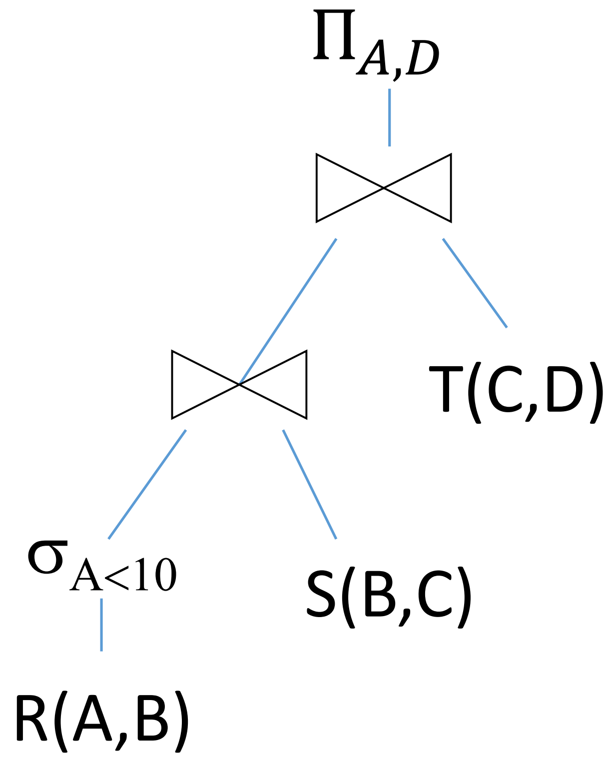
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$

Push down selection on A so it occurs earlier



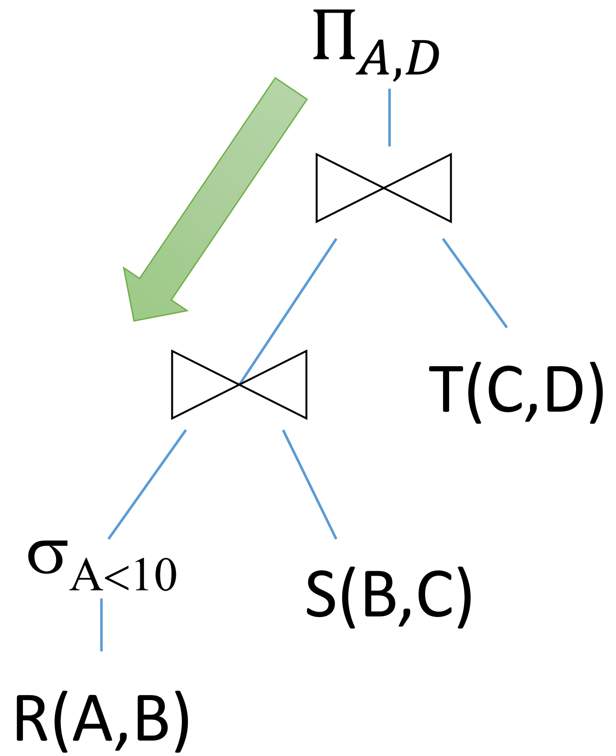
Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))$$

Push down projection so it occurs earlier



Optimizing RA Plan

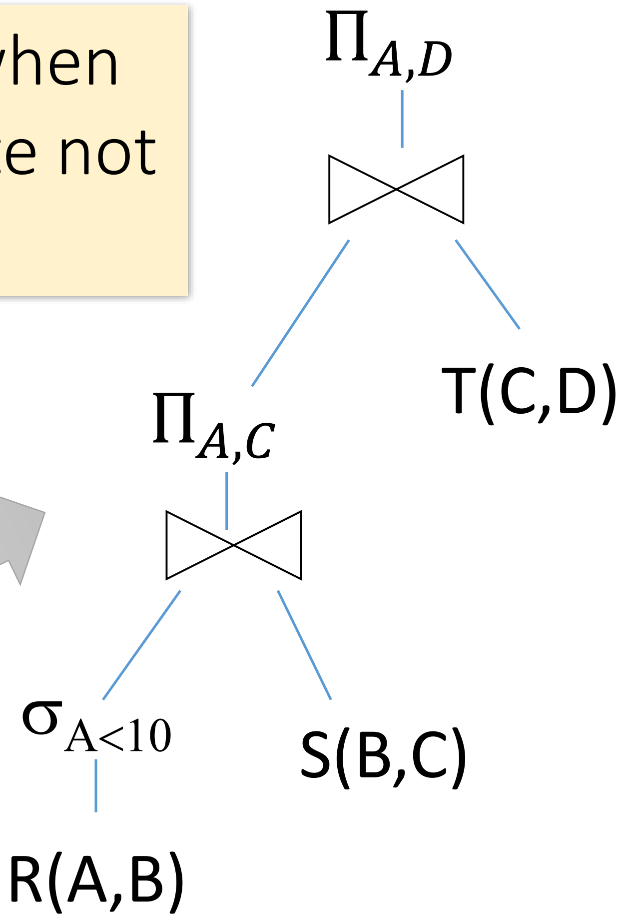
R(A,B) S(B,C) T(C,D)

SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;

$$\Pi_{A,D}\left(T \bowtie \Pi_{A,c}(\sigma_{A<10}(R) \bowtie S)\right)$$

We eliminate B earlier!

In general, when is an attribute not needed…?



3. Transactions and ACID

Transactions: Basic Definition

A <u>**transaction ("TXN")**</u> is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
    UPDATE Product
    SET Price = Price – 1.99
    WHERE pname = 'Gizmo'
COMMIT
```

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects *a single real-world transition*.

In the real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’
COMMIT
```

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

ACID: Atomicity

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*

- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

Transactions

- A key concept is the **transaction (TXN)**: an **atomic** sequence of db actions (reads/writes)

<u>Atomicity</u>: An action either completes *entirely* or *not at all*

Acct	Balance
a10	20,000
a20	15,000

Transfer $3k from a10 to a20:
1. Debit $3k from a10
2. Credit $3k to a20

Acct	Balance
a10	17,000
a20	18,000

Written naively, in which states is **atomicity** preserved?

- Crash before 1,
- After 1 but before 2,
- After 2.

DB Always preserves atomicity!

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can’t be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions
- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
    - E.g. Should not be able to observe changes from other transactions during the run

Challenge: Scheduling Concurrent Transactions

- The DBMS ensures that the execution of $\{T_1,\ldots,T_n\}$ is equivalent to some **serial** execution
- One way to accomplish this: **Locking**
  - Before reading or writing, transaction requires a lock from DBMS, holds until the end
- **Key Idea**: If T_i wants to write to an item x and T_j wants to read x, then T_i, T_j **conflict**. Solution via locking:
  - only one winner gets the lock
  - loser is blocked (waits) until winner finishes

A set of TXNs is <u>**isolated**</u> if their effect is as if all were executed serially

What if T_i and T_j need X and Y, and T_i asks for X before T_j, and T_j asks for Y before T_i?
-> *Deadlock!* One is aborted…

All concurrency issues handled by the DBMS…

ACID: Durability

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to **disk**

Ensuring Atomicity & Durability

- DBMS ensures **atomicity** even if a TXN crashes!

- One way to accomplish this: **Write-ahead logging (WAL)**

- **Key Idea**: Keep a log of all the writes done.
  - After a crash, the partially executed TXNs are undone using the log

Write-ahead Logging (WAL): Before any action is finalized, a corresponding log entry is forced to disk

We assume that the log is on "stable" storage

All atomicity issues also handled by the DBMS...

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures

- Users may abort the program: need to "rollback the changes"
  - Need to *log* what happened

- Many users executing concurrently
  - Can be solved via locking (we'll see this next lecture!)

And all this with… Performance!!

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**

- Many newer “NoSQL” DBMSs relax ACID

- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

ACID is an extremely important & successful paradigm, but still debated!

Summary of DBMS

- DBMS are used to maintain, query, and manage large datasets.
  - Provide concurrency, recovery from crashes, quick application development, integrity, and security

- Key abstractions give **data independence**

- DBMS R&D is one of the broadest fields in CS. **Fact!**