

SOUG Day Spring, 17th April 2024

# Busting some database-myths

episode 1



Erich Steiger  
Lead Database Architect

18/04/2024

Erich Steiger



# Myths about me

- Lead Database Architect
  - 3.5 years at TWINT
- Software Engineering Background
  - Java / C++ and other programming languages
  - 27 years of Application development experience
  - Over 20 years in the financial sector
- Database knowledge
  - 27 years of Database experience
  - 21 years of Oracle experience
- First Program sold at age 15
- Unix since 1995 / Linux since 1993
- Private
  - 45 years old
  - Show-Jumping
  - Skiing

**CONFIRMED**

**PLAUSIBLE**

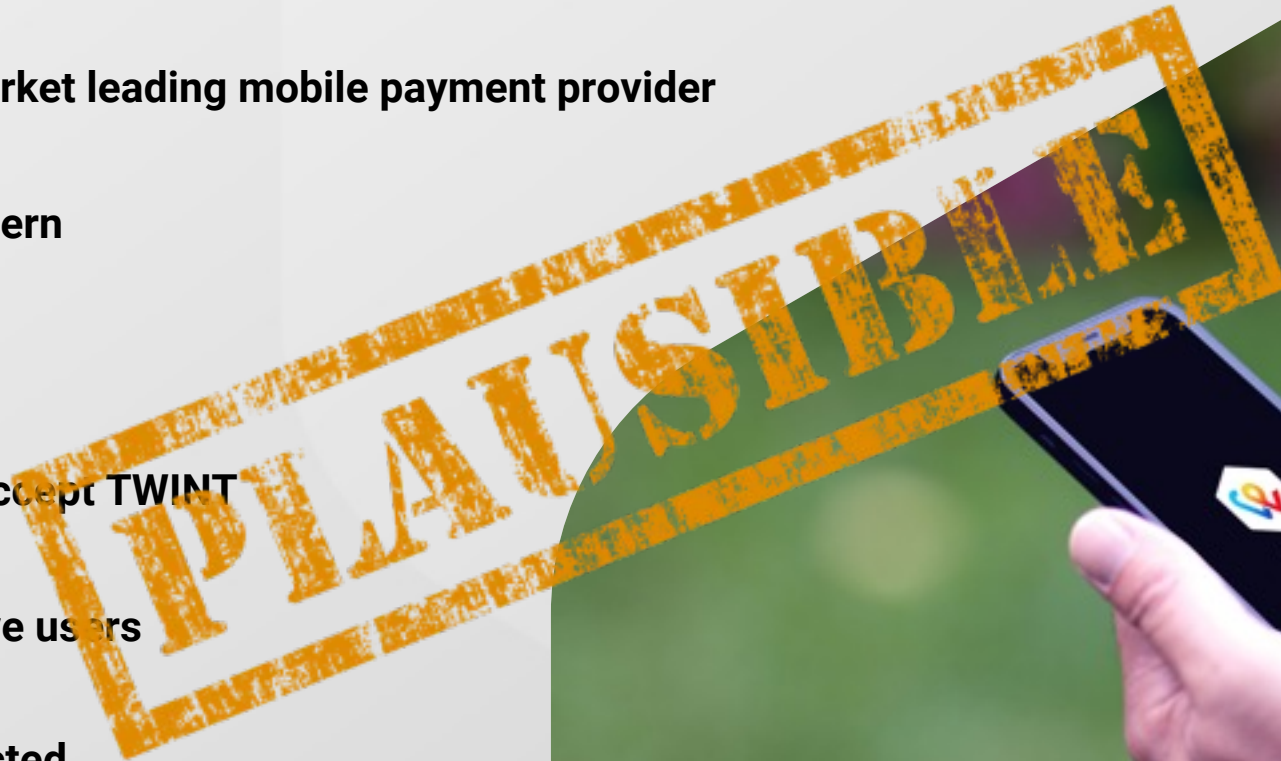
**CONFIRMED**



# Myths about TWINT AG

**TWINT is Switzerland's market leading mobile payment provider**

- **Located in Zürich and Bern**
- **210 Employees**
- **Around 70% of shops accept TWINT**
- **More than 5.5 Mio active users**
- **38 Issuer Banks connected**





# Who heard about the following Myths?



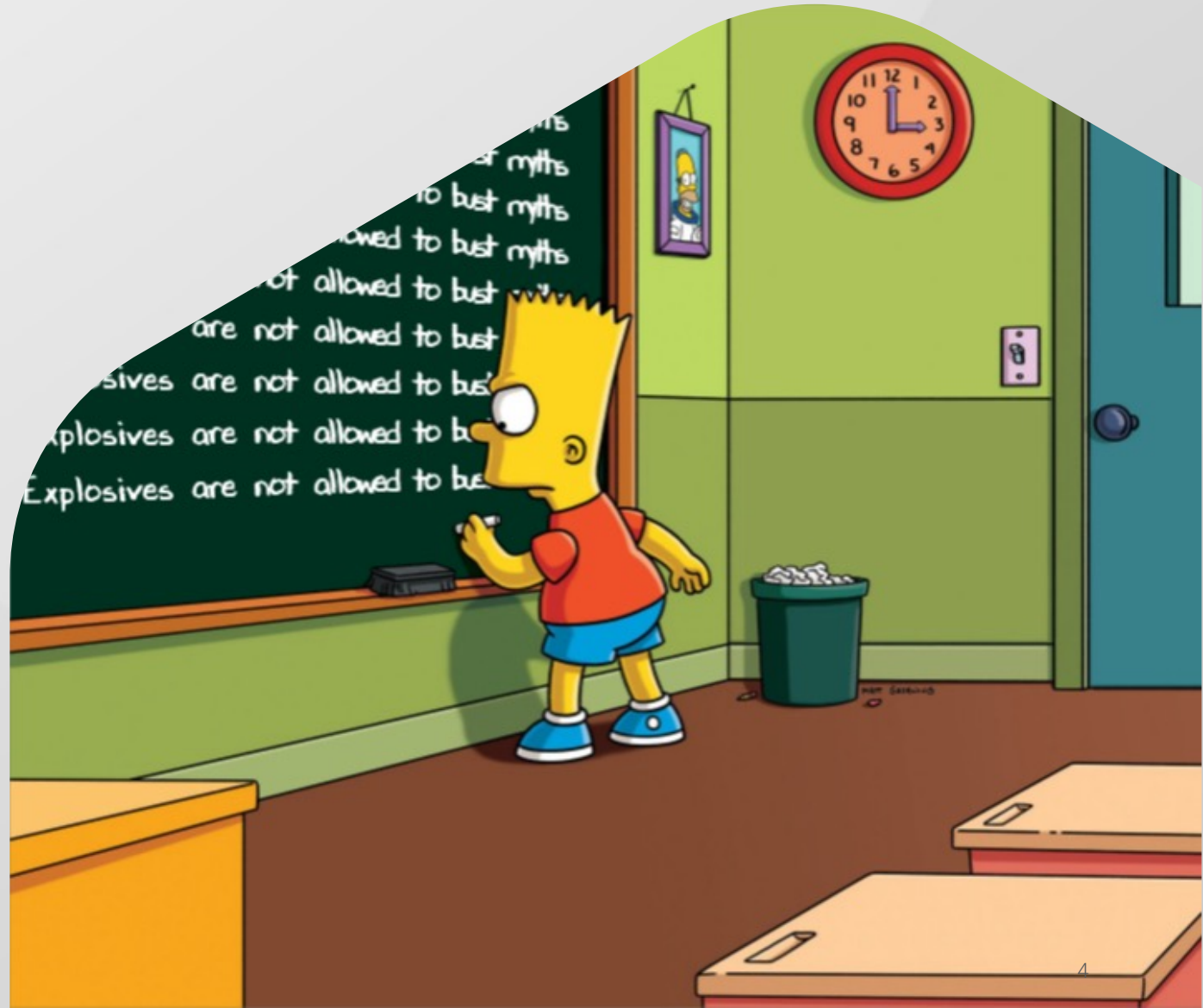
NoSQL is faster than relational DB



Oracle is the most expensive DB

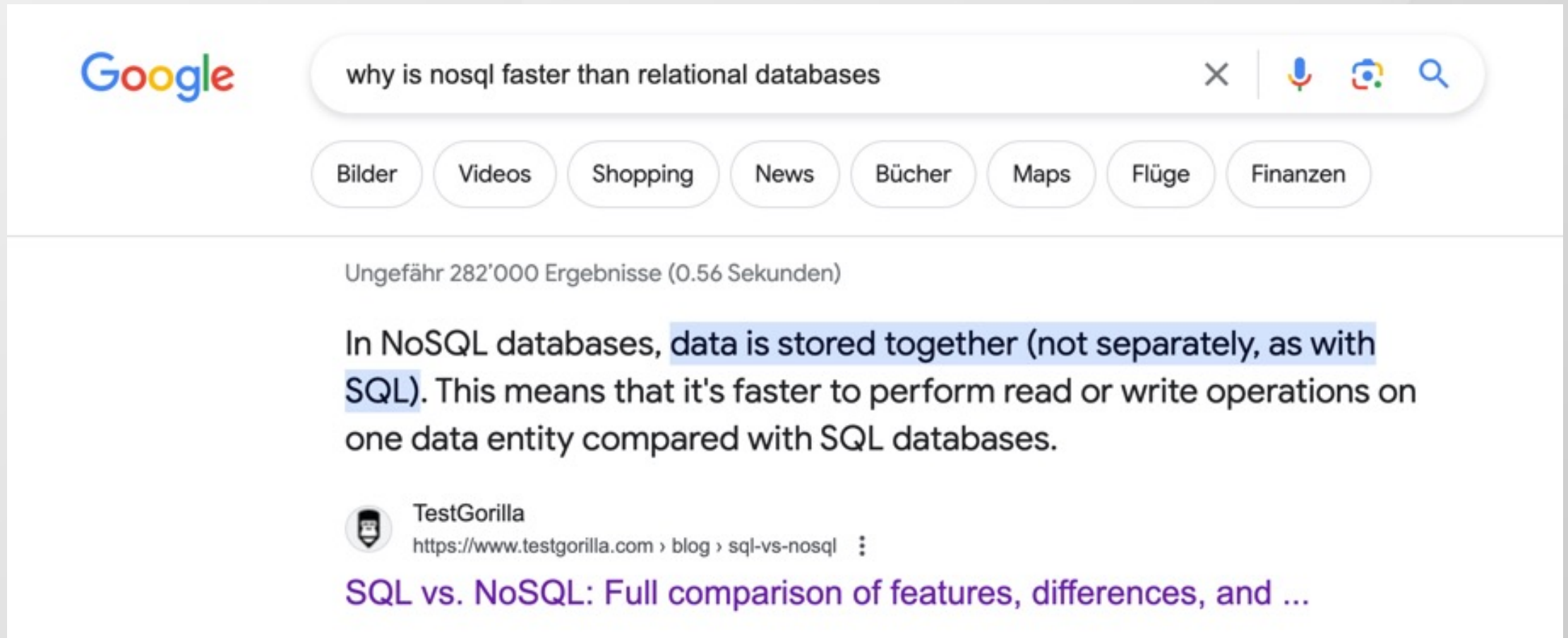


AWS DynamoDB is ACID-compliant



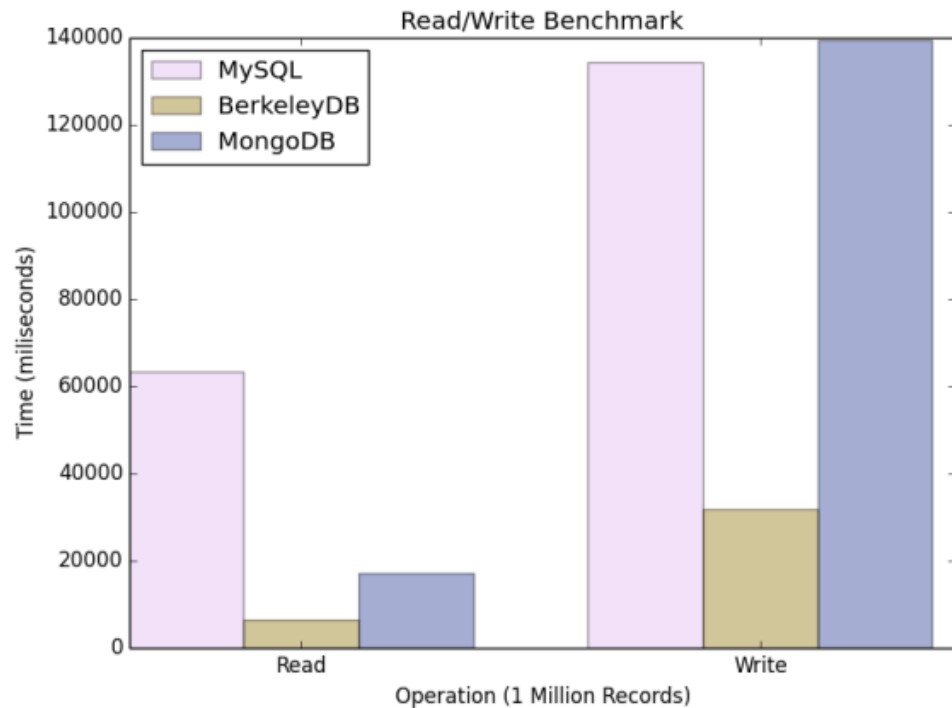
**NoSQL is faster than relational DB**

# Ask Google

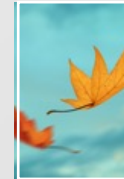


The image shows a Google search interface. At the top left is the Google logo. The search bar contains the text "why is nosql faster than relational databases". To the right of the search bar are icons for clearing the search, voice search, image search, and a magnifying glass. Below the search bar are several filter buttons: "Bilder", "Videos", "Shopping", "News", "Bücher", "Maps", "Flüge", and "Finanzen". Below the filters, it says "Ungefähr 282'000 Ergebnisse (0.56 Sekunden)". The main search result is from "TestGorilla" with the URL "https://www.testgorilla.com > blog > sql-vs-nosql". The title of the result is "SQL vs. NoSQL: Full comparison of features, differences, and ...". The main text of the result states: "In NoSQL databases, data is stored together (not separately, as with SQL). This means that it's faster to perform read or write operations on one data entity compared with SQL databases."

# Ruihan Wang, Zongyan Yang - University of Rochester



**Figure 3.** read/write time (less is better)



Paper from Fall 2017



Same Hardware



BerkeleyDB beats  
MongoDB read and  
write



MySQL beats  
MongoDB on write

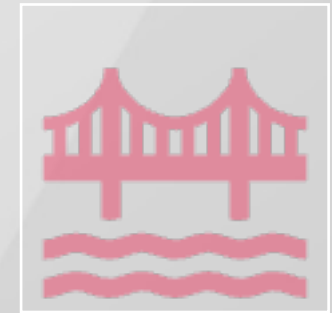
# Irrelevant in the Cloud



Hardware



Underlying Technology Stack



Architecture



# Relevant in the Cloud



Performance and  
Latency



Throughput



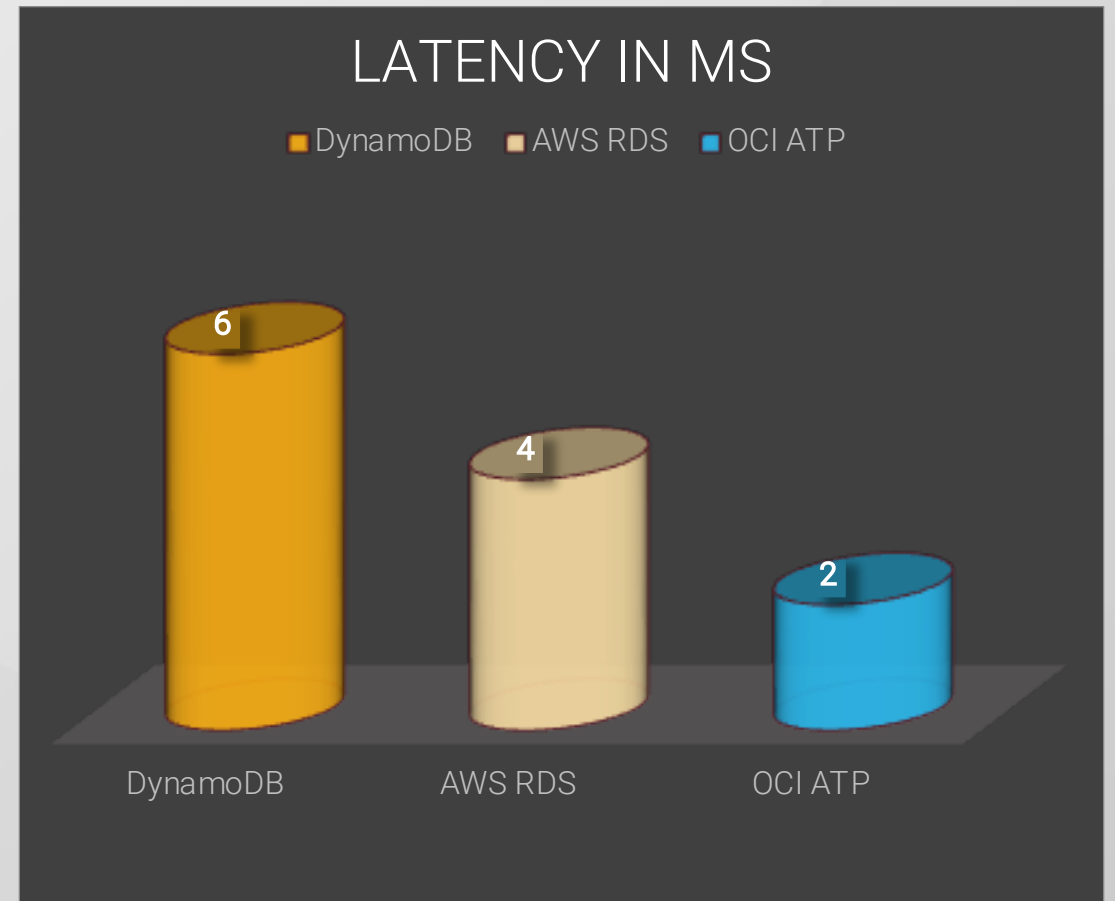
Price




Availability

# RDBMS is faster than NoSQL

- Benchmark-Application inserts Business-Objects
- On Relational DBs a Business-Objects consists of 1 Main-Record and 2 Child-Records
- 22 indexes directly involved
- 5 FK constraints directly involved
- On DynamoDB one Business-Object is one record
- PK index
- SK index
- JSON Object instead of relational storage



# Maybe lack of Knowledge?

 **Alex DeBrie** ✓  
@alexdbrie

Been using a relational database for a project recently. First time in a while that I've been using it for OLTP. Some quick reflections.

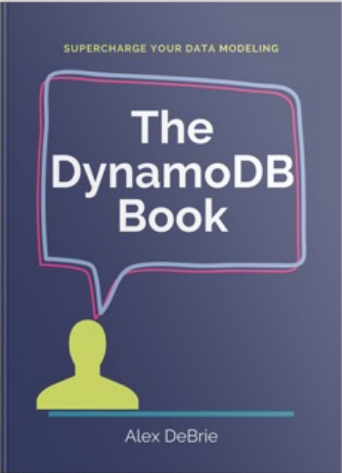
The good:

- The query flexibility! Ohh, the query flexibility. How I've missed thee. I really do appreciate that.

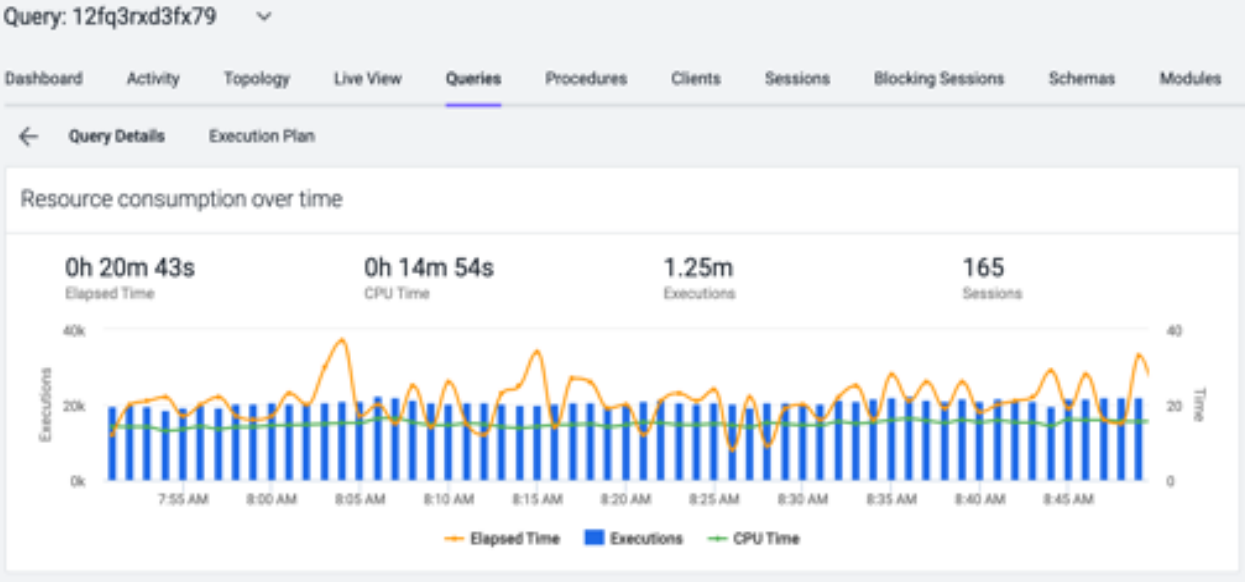
The less good:

- All the schema stuff is so annoying when iterating. Constantly going back + forth between doing it the right way and doing it the fast way.
- I get a tiny pit in my stomach not knowing what's happening on a given query and how it will work with more data. Especially true when you start having joins across 5 tables with some filter conditions.

2:26 PM · Mar 21, 2024 · 18.1K Views



```
SELECT
  a.field1,
  b.field2,
  ...
FROM
  table_a a
  LEFT OUTER JOIN table_b b ON a.some_uuid = b.uuid
  LEFT OUTER JOIN table_c c ON a.uuid = c.table_a_uuid
  LEFT OUTER JOIN table_d d ON a.uuid = d.table_a_uuid
  LEFT OUTER JOIN table_e e ON a.uuid = e.table_a_uuid
  LEFT OUTER JOIN table_f f ON f.receiver_uuid = e.uuid
  LEFT OUTER JOIN table_g g ON g.sender_uuid = f.uuid
WHERE
  a.uuid = :1
```



**NoSQL is faster than relational DB**



**Oracle is the most expensive DB**



# DEMO



**Warning:** the following figures are taken from cloud cost calculators between February and April 2024 in CHF. Please, be aware, that cloud prices and currency exchange rates may have changed since then. Also, the cost calculator software might have changed since then, sometimes on a daily base. The prices might not be 100% accurate anymore.

# AWS DB Cost estimate – smallest shapes

[AWS Pricing Calculator](#) > My Estimate

**My Estimate** [Edit](#) [↗](#) [Export](#) [Share](#)

### Estimate summary [Info](#)

|              |              |                                      |
|--------------|--------------|--------------------------------------|
| Upfront cost | Monthly cost | Total 12 months cost                 |
| 0.00 USD     | 433.80 USD   | <b>5,205.60 USD</b>                  |
|              |              | <small>Includes upfront cost</small> |

### Getting Started with AWS

[Get started for free](#) [Contact Sales](#)

### My Estimate

[Duplicate](#) [Delete](#) [Move to](#) [Create group](#) [Add support](#) [Add service](#)

| <input type="checkbox"/> | Service Name              | Status | Upfron... | Monthly ... | Descrip... | Region            | Config Summary                                                                                                                                     |
|--------------------------|---------------------------|--------|-----------|-------------|------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <input type="checkbox"/> | Amazon DynamoDB           | ↗      | 0.00 USD  | 6.12 USD    | -          | Europe (Frankf... | Table class (Standard), Average item size (all attributes) (1 KB), Data storage size (20 GB)                                                       |
| <input type="checkbox"/> | Amazon RDS for SQL server | ↗      | 0.00 USD  | 57.57 USD   | -          | Europe (Frankf... | Storage for each RDS instance (General Purpose SSD (gp2)), Storage amount (20 GB), Number of nodes (1), Instance type (db.t2.micro), Utilizatio... |
| <input type="checkbox"/> | Amazon RDS for MySQL      | ↗      | 0.00 USD  | 71.24 USD   | -          | Europe (Frankf... | Storage for each RDS instance (General Purpose SSD (gp2)), Storage amount (20 GB), Quantity (1), Instance type (db.t2.micro), Utilization (On-D... |
| <input type="checkbox"/> | Amazon RDS for MariaDB    | ↗      | 0.00 USD  | 71.26 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp3)), Storage amount (20 GB), Quantity (1), Instance type (db.t2.micro), Utilization (On-Demand only) (...   |
| <input type="checkbox"/> | Amazon RDS for PostgreSQL | ↗      | 0.00 USD  | 74.89 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp2)), Storage amount (20 GB), Nodes (1), Instance Type (db.t2.micro), Utilization (On-Demand only) (10...    |
| <input type="checkbox"/> | Amazon RDS for Oracle     | ↗      | 0.00 USD  | 75.62 USD   | -          | Europe (Frankf... | Storage amount (20 GB), Storage for each RDS instance (General Purpose SSD (gp2)), Number of RDS for Oracle instances (1), Instance type (db...    |
| <input type="checkbox"/> | Amazon RDS for Db2        | ↗      | 0.00 USD  | 77.10 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp3)), Storage amount (20 GB), Nodes (1), Instance Type (db.t3.small), Utilization (On-Demand only) (10...    |

<https://calculator.aws/-/estimate>

# Azure DB estimate – smallest shapes





## Azure Small Shapes



|                                 |  |                                                         |  |                   |                     |
|---------------------------------|--|---------------------------------------------------------|--|-------------------|---------------------|
| ▼ Azure SQL Database            |  | Single Database, vCore, General Purpose, Provision...   |  | Upfront: CHF 0.00 | Monthly: CHF 378.40 |
| ▼ Azure Database for MySQL      |  | Single Server Deployment, Basic Tier, 1 Gen 5 (2 vC...  |  | Upfront: CHF 0.00 | Monthly: CHF 71.89  |
| ▼ Azure Database for PostgreSQL |  | Single Server Deployment, Basic Tier, 1 Gen 5 (2 vC...  |  | Upfront: CHF 0.00 | Monthly: CHF 71.89  |
| ▼ Azure Database for MariaDB    |  | Basic Tier, 1 Gen 5 (2 vCore) x 730 Hours, 20 GB Sto... |  | Upfront: CHF 0.00 | Monthly: CHF 71.89  |
| ▼ Azure Cosmos DB               |  | Azure Cosmos DB for NoSQL (formerly Core), Stand...     |  | Upfront: CHF 0.00 | Monthly: CHF 36.14  |

<https://azure.microsoft.com/en-us/pricing/calculator/> 17.4.2024

# GCP DB estimate – medium shapes

|                                                                                                             |            |   |
|-------------------------------------------------------------------------------------------------------------|------------|---|
|  MySQL<br>Cloud SQL        | CHF 21.57  | ⋮ |
|  MySQL 2<br>Cloud SQL      | CHF 21.57  | ⋮ |
|  Firestore                 | CHF 3.96   | ⋮ |
|  SQL Server<br>Cloud SQL | CHF 416.04 | ⋮ |

<https://www.oracle.com/ch-de/cloud/costestimator.html> 17.4.2024

# OCI DB estimate – smallest shapes

**Meine Schätzung** ✎ ⋮ Gratis starten CHF - Swiss Fran ▾ Geschätzte Monatliche Kosten  
Kosten für OCI-Services konfigurieren und schätzen ([Weitere Informationen](#)) **CHF 729.13** 📄

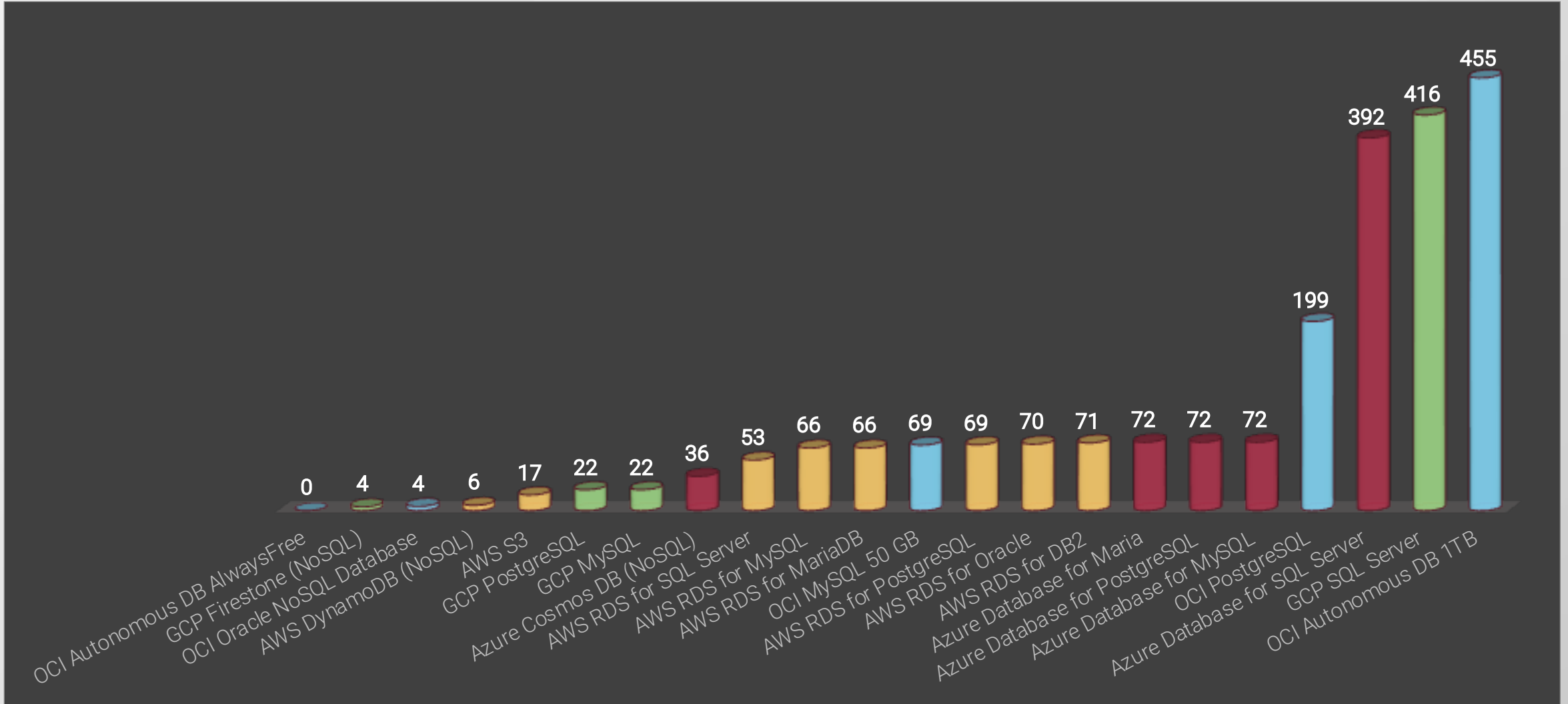
Konfiguration hinzufügen

|                                                  |                                                  |
|--------------------------------------------------|--------------------------------------------------|
| <b>Autonomous Database</b> ⋮                     | Geschätzte Monatliche Kosten <b>CHF 455.61</b> > |
| <b>MySQL Database Service</b> ⋮                  | Geschätzte Monatliche Kosten <b>CHF 69.35</b> >  |
| <b>Oracle NoSQL Database Cloud - On-Demand</b> ⋮ | Geschätzte Monatliche Kosten <b>CHF 4.49</b> >   |
| <b>Database with PostgreSQL</b> ⋮                | Geschätzte Monatliche Kosten <b>CHF 199.67</b> > |

<https://www.oracle.com/ch-de/cloud/costestimator.html> 16.4.2023



# Overview – smallest shapes 20 GB



# AWS DB Cost estimate – medium shapes

AWS Pricing Calculator > My Estimate

## My Estimate [Edit](#)

[Export](#) [Share](#)

### Estimate summary [Info](#)

|              |          |              |              |                      |                       |
|--------------|----------|--------------|--------------|----------------------|-----------------------|
| Upfront cost | 0.00 USD | Monthly cost | 6,238.18 USD | Total 12 months cost | <b>74,858.16 USD</b>  |
|              |          |              |              |                      | Includes upfront cost |

### Getting Started with AWS

[Get started for free](#) [Contact Sales](#)

### My Estimate

[Duplicate](#) [Delete](#) [Move to](#) [Create group](#) [Add support](#) [Add service](#)





< 1 >
















| <input type="checkbox"/> | Service Name              | Status | Upfron... | Monthly ...  | Descrip... | Region            | Config Summary                                                                                                                                     |
|--------------------------|---------------------------|--------|-----------|--------------|------------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <input type="checkbox"/> | Amazon RDS for SQL server | -      | 0.00 USD  | 712.18 USD   | -          | Europe (Frankf... | Storage for each RDS instance (General Purpose SSD (gp2)), Storage amount (1 TB), Number of nodes (1), Instance type (db.t2.micro), Utilization... |
| <input type="checkbox"/> | Amazon RDS for MySQL      | -      | 0.00 USD  | 863.42 USD   | -          | Europe (Frankf... | Storage for each RDS instance (General Purpose SSD (gp3)), Storage amount (1 TB), Quantity (1), Instance type (db.t2.micro), Utilization (On-De... |
| <input type="checkbox"/> | Amazon RDS for MariaDB    | -      | 0.00 USD  | 863.42 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp3)), Storage amount (1 TB), Quantity (1), instance type (db.t2.micro), Utilization (On-Demand only) (10...  |
| <input type="checkbox"/> | Amazon RDS for PostgreSQL | -      | 0.00 USD  | 867.07 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp3)), Storage amount (1 TB), Nodes (1), Instance Type (db.t2.micro), Utilization (On-Demand only) (100 ...   |
| <input type="checkbox"/> | Amazon RDS for Oracle     | -      | 0.00 USD  | 867.80 USD   | -          | Europe (Frankf... | Storage amount (1 TB), Storage for each RDS instance (General Purpose SSD (gp3)), Number of RDS for Oracle instances (1), instance type (db.t2...  |
| <input type="checkbox"/> | Amazon RDS for Db2        | -      | 0.00 USD  | 869.26 USD   | -          | Europe (Frankf... | Storage volume (General Purpose SSD (gp3)), Storage amount (1 TB), Nodes (1), Instance Type (db.t3.small), Utilization (On-Demand only) (100 ...   |
| <input type="checkbox"/> | Amazon DynamoDB           | -      | 0.00 USD  | 1,195.03 USD | -          | Europe (Frankf... | Table class (Standard), Average item size (all attributes) (1 KB), Data storage size (1 TB)                                                        |

# Azure DB estimate – medium shapes




Azure Small Shapes +

### Azure Small Shapes



|                                 |                                                                                     |                                                        |                                                                                                                                                                             |                 |                   |
|---------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|-------------------|
| ✓ Azure Cosmos DB               |    | Azure Cosmos DB for NoSQL (formerly Core), Stand...    |       | Upfront: \$0.00 | Monthly: \$484.16 |
| ✓ Azure SQL Database            |    | Single Database, vCore, General Purpose, Provision...  |       | Upfront: \$0.00 | Monthly: \$659.35 |
| ✓ Azure Database for MySQL      |    | Single Server Deployment, General Purpose Tier, 1 ...  |       | Upfront: \$0.00 | Monthly: \$991.76 |
| ✓ Azure Database for MariaDB    |  | General Purpose Tier, 1 Gen 5 (2 vCore) x 730 Hours... |   | Upfront: \$0.00 | Monthly: \$757.66 |
| ✓ Azure Database for PostgreSQL |  | Single Server Deployment, General Purpose Tier, 1 ...  |   | Upfront: \$0.00 | Monthly: \$757.66 |

# GCP DB estimate – medium shapes

|                                                                                     |                                |            |   |
|-------------------------------------------------------------------------------------|--------------------------------|------------|---|
|    | <b>PostgreSQL</b><br>Cloud SQL | CHF 608.53 | ⋮ |
|    | <b>MySQL</b><br>Cloud SQL      | CHF 608.53 | ⋮ |
|  | <b>SQL Server</b><br>Cloud SQL | CHF 984.30 | ⋮ |

# OCI DB estimate – medium shapes

## Meine Schätzung ...

Kosten für OCI-Services konfigurieren und schätzen ([Weitere Informationen](#))

Gratis starten

CHF - Swiss Fran 

Geschätzte Monatliche Kosten

**CHF 1'812.88** 

Konfiguration hinzufügen

**Oracle NoSQL Database Cloud - On-Demand ...**

Geschätzte Monatliche Kosten **CHF 358.77** >

**Database with PostgreSQL ...**

Geschätzte Monatliche Kosten **CHF 311.15** >

**MySQL Database Service ...**

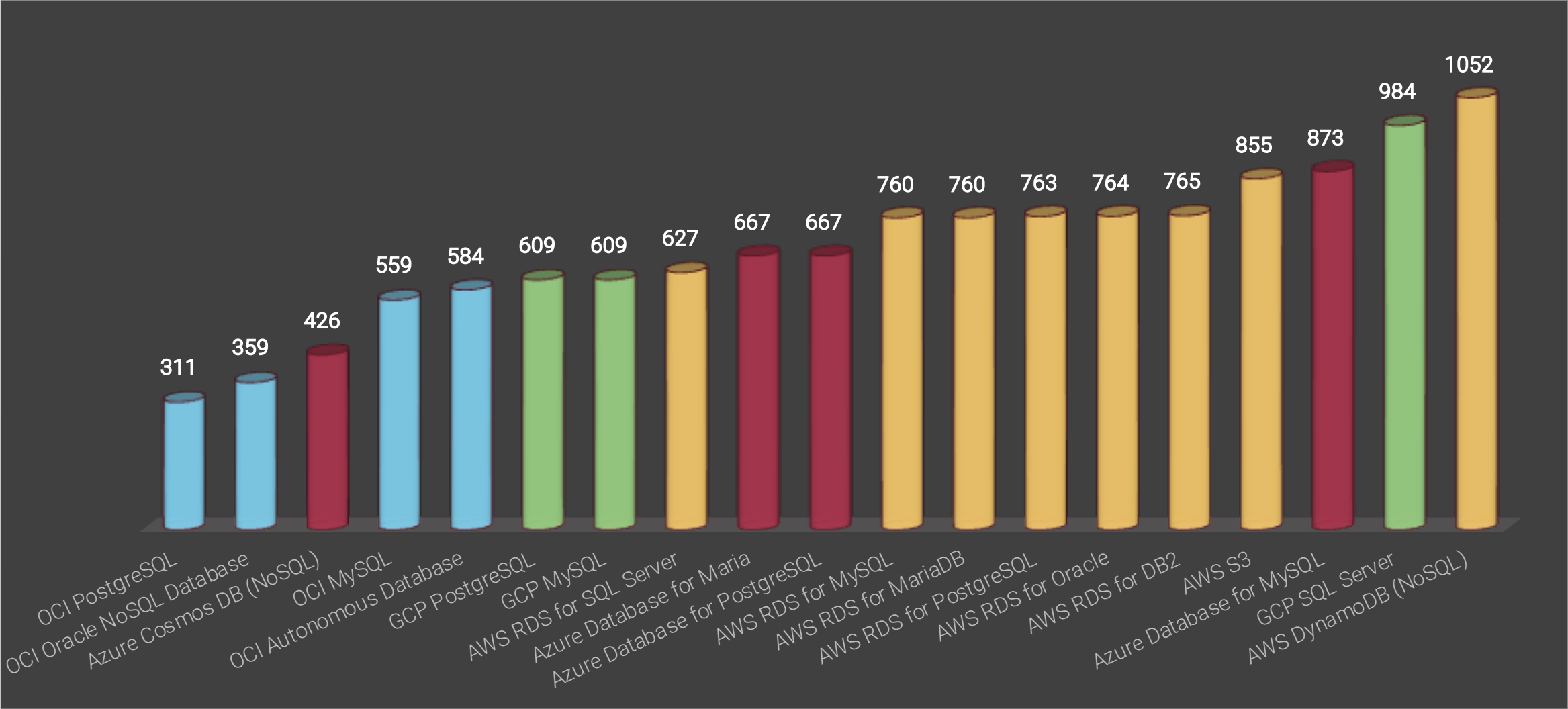
Geschätzte Monatliche Kosten **CHF 558.95** >

**Autonomous Database ...**

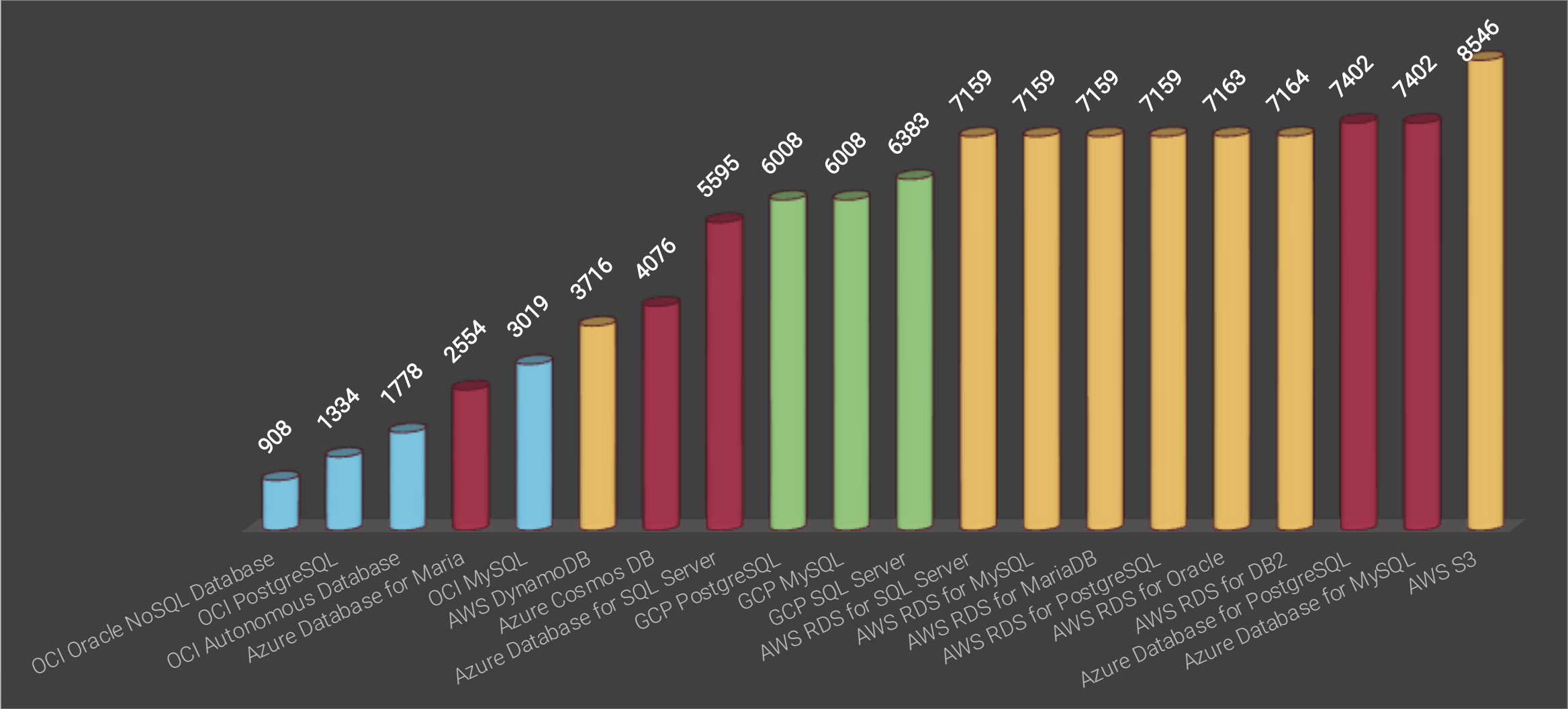
Geschätzte Monatliche Kosten **CHF 584.01** >



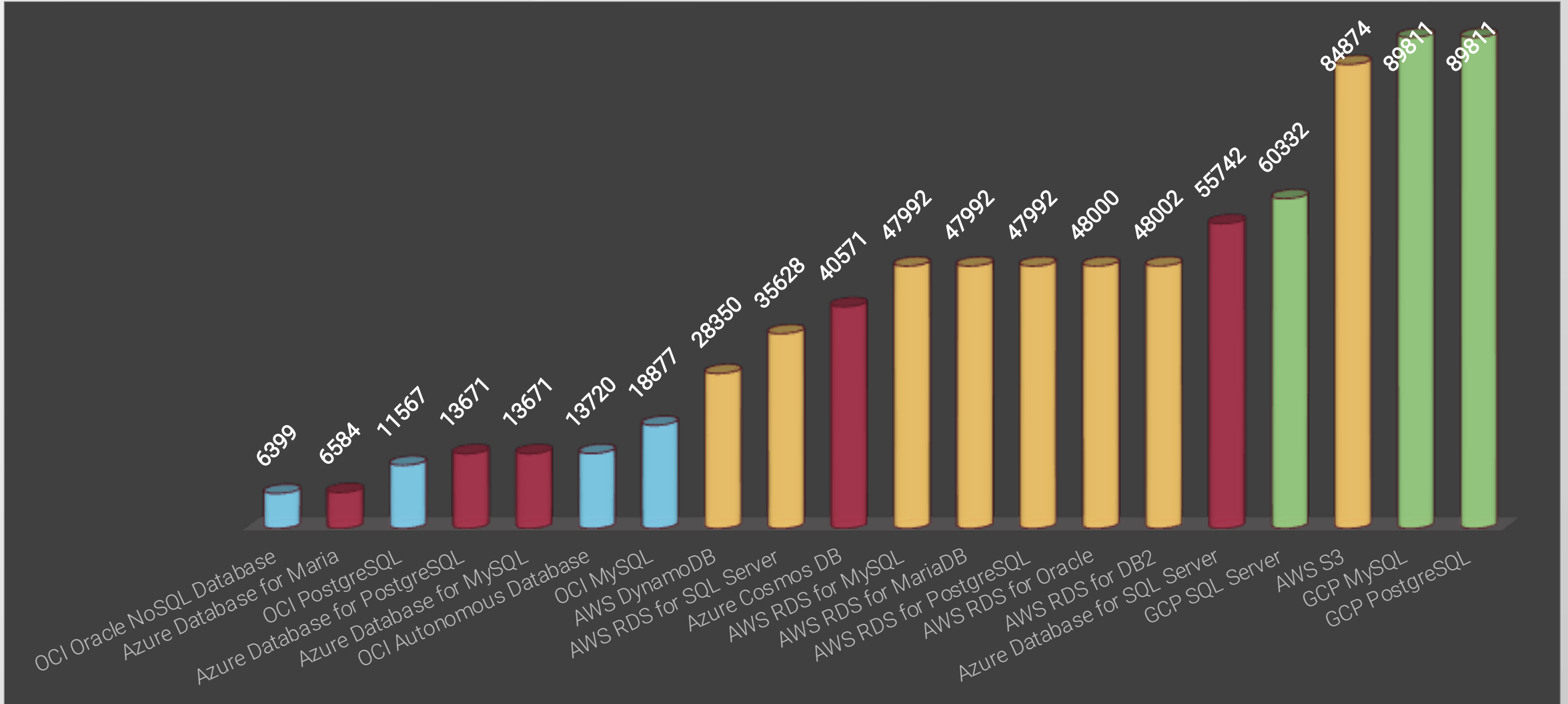
# Overview – medium shapes 1 TB



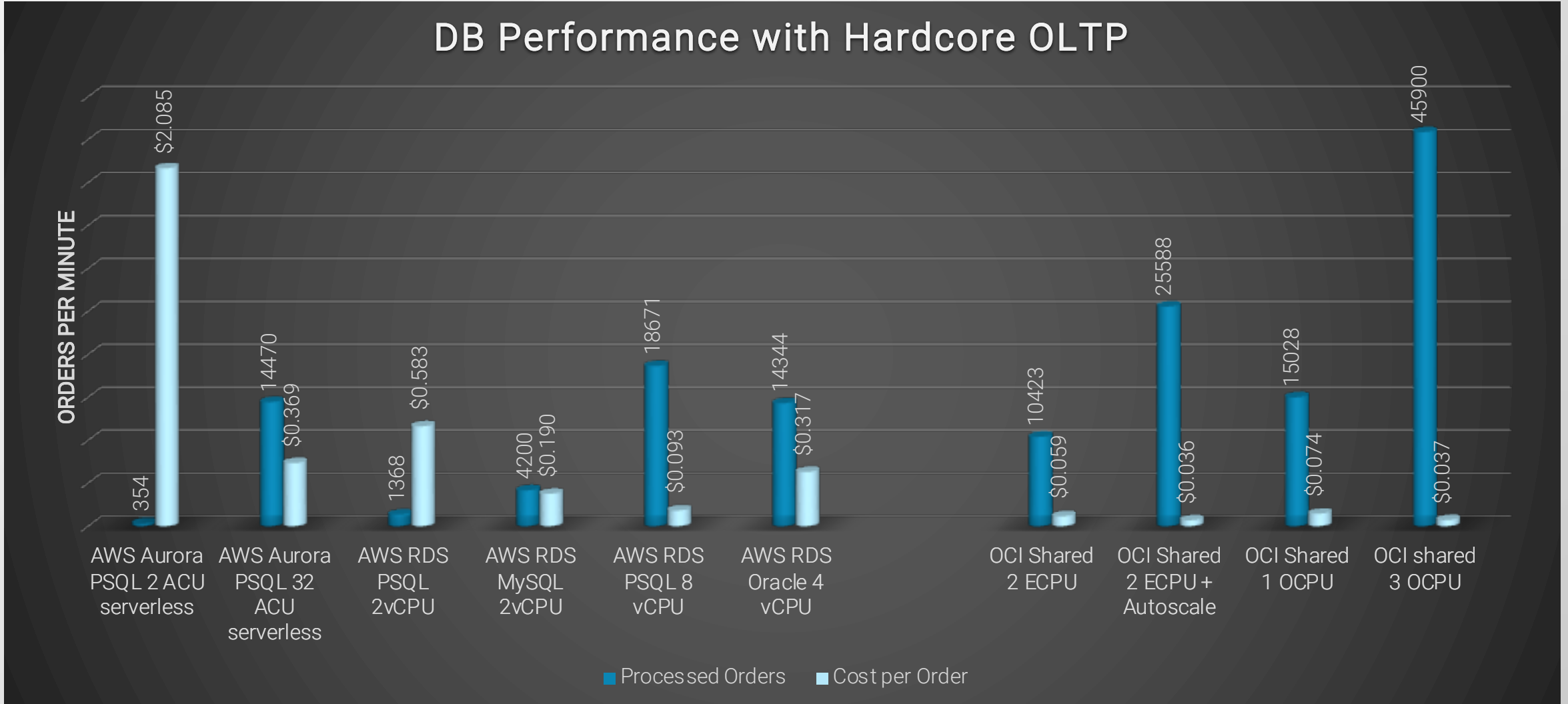
# Overview – large shapes 10 TB



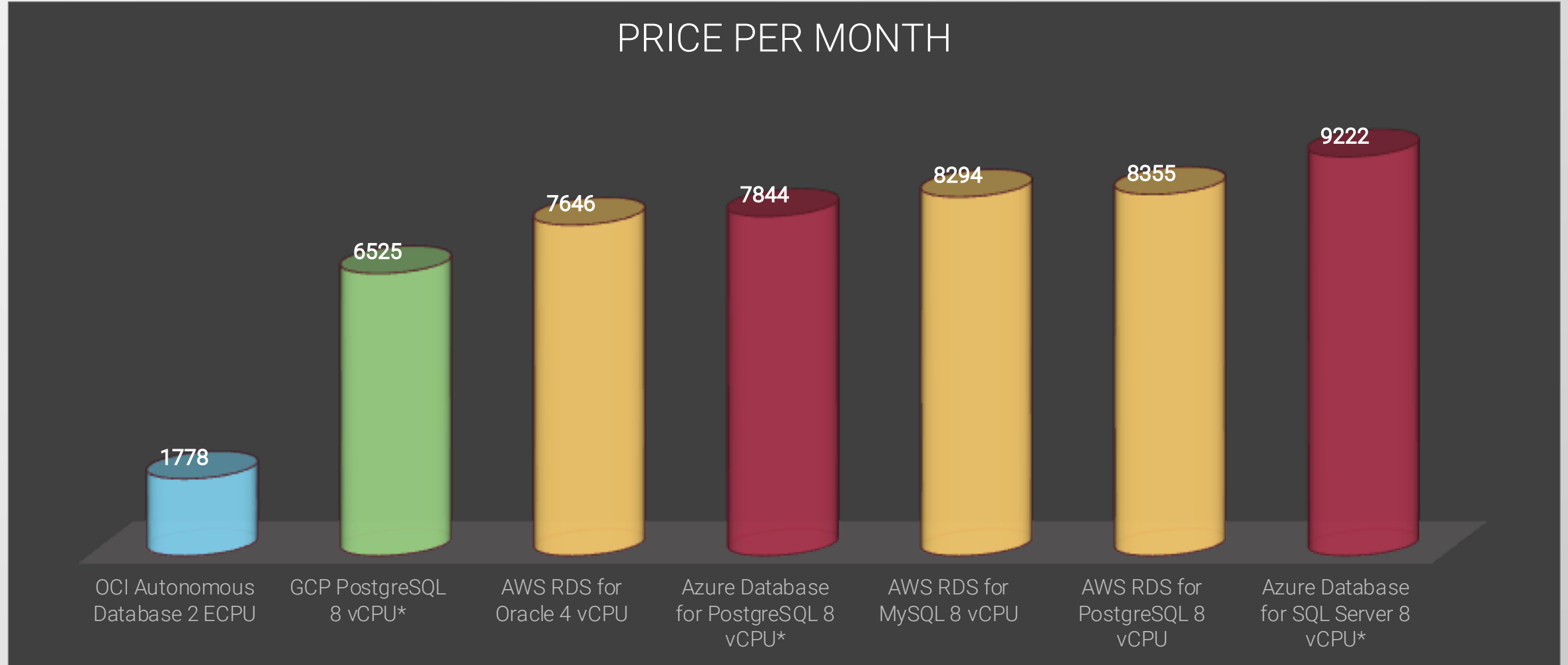
# Overview – x-large shapes 100 TB



# Throughput and price per Order



# Databases able to process 10k orders per minute



\* = no benchmark results taken, estimated size is guessed by results of AWS services and reality might look different

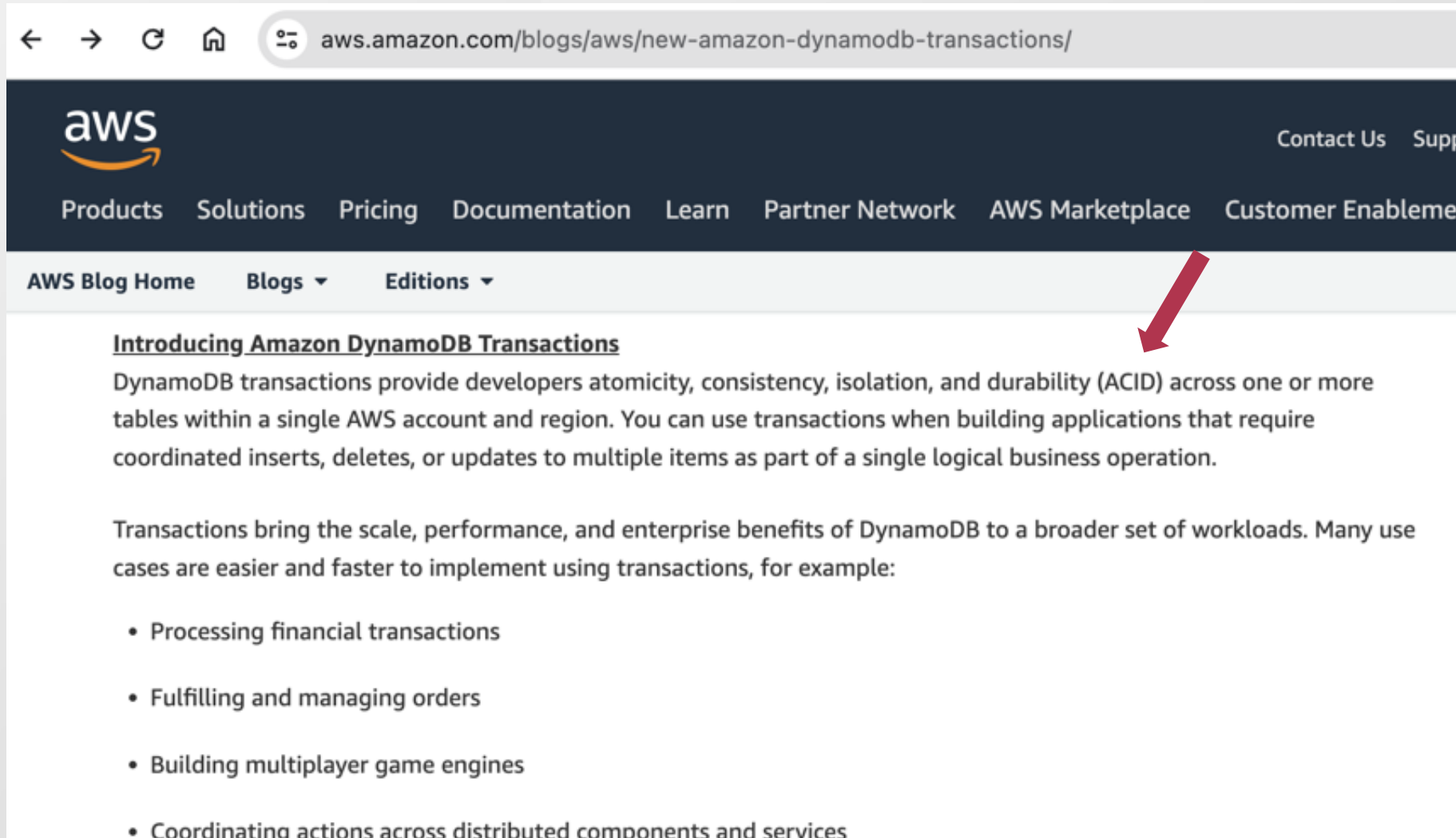
**Oracle is the most expensive DB**



**AWS DynamoDB is ACID-compliant**



# AWS says, DynamoDB is ACID compliant



The screenshot shows the AWS blog page for the article 'Introducing Amazon DynamoDB Transactions'. The URL in the browser is [aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/](https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/). The AWS logo is in the top left, and navigation links like 'Products', 'Solutions', 'Pricing', 'Documentation', 'Learn', 'Partner Network', 'AWS Marketplace', and 'Customer Enablement' are in the top right. Below the navigation, there are links for 'AWS Blog Home', 'Blogs', and 'Editions'. The main content area features the article title 'Introducing Amazon DynamoDB Transactions' and a paragraph stating: 'DynamoDB transactions provide developers atomicity, consistency, isolation, and durability (ACID) across one or more tables within a single AWS account and region. You can use transactions when building applications that require coordinated inserts, deletes, or updates to multiple items as part of a single logical business operation.' A red arrow points to the word 'ACID' in this paragraph. Below the paragraph, there is another paragraph: 'Transactions bring the scale, performance, and enterprise benefits of DynamoDB to a broader set of workloads. Many use cases are easier and faster to implement using transactions, for example:' followed by a bulleted list of use cases.

**Introducing Amazon DynamoDB Transactions**

DynamoDB transactions provide developers atomicity, consistency, isolation, and durability (ACID) across one or more tables within a single AWS account and region. You can use transactions when building applications that require coordinated inserts, deletes, or updates to multiple items as part of a single logical business operation.

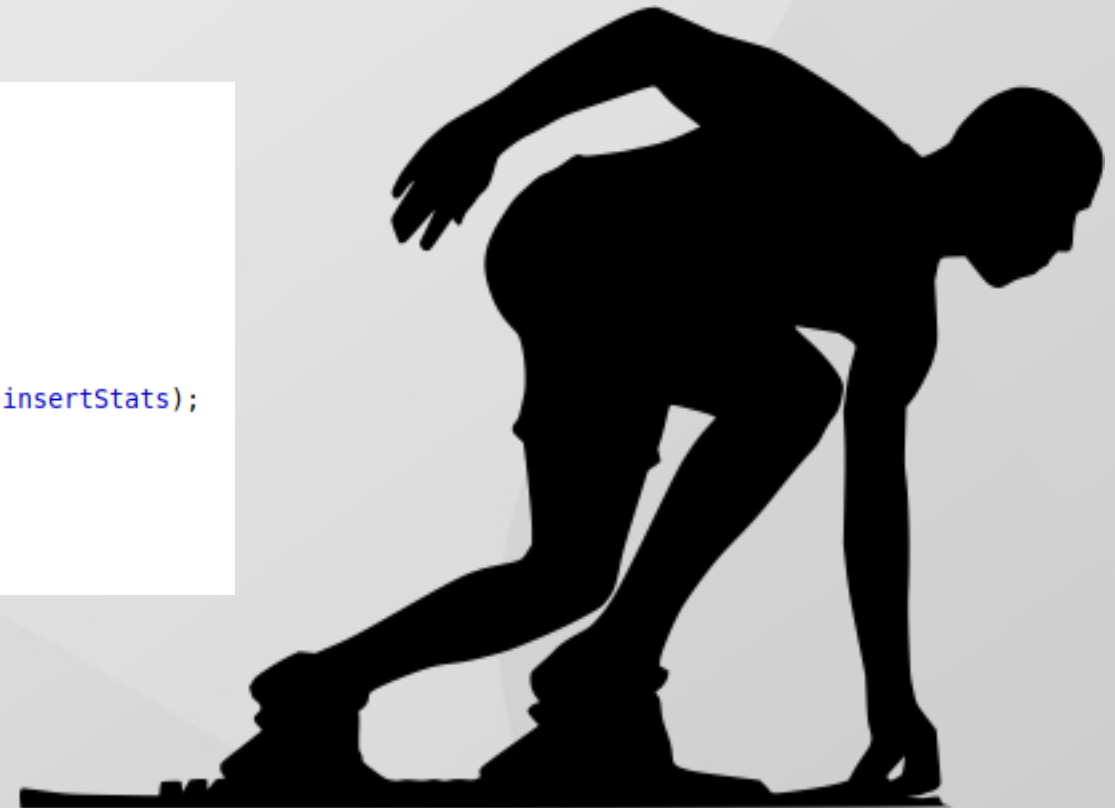
Transactions bring the scale, performance, and enterprise benefits of DynamoDB to a broader set of workloads. Many use cases are easier and faster to implement using transactions, for example:

- Processing financial transactions
- Fulfilling and managing orders
- Building multiplayer game engines
- Coordinating actions across distributed components and services

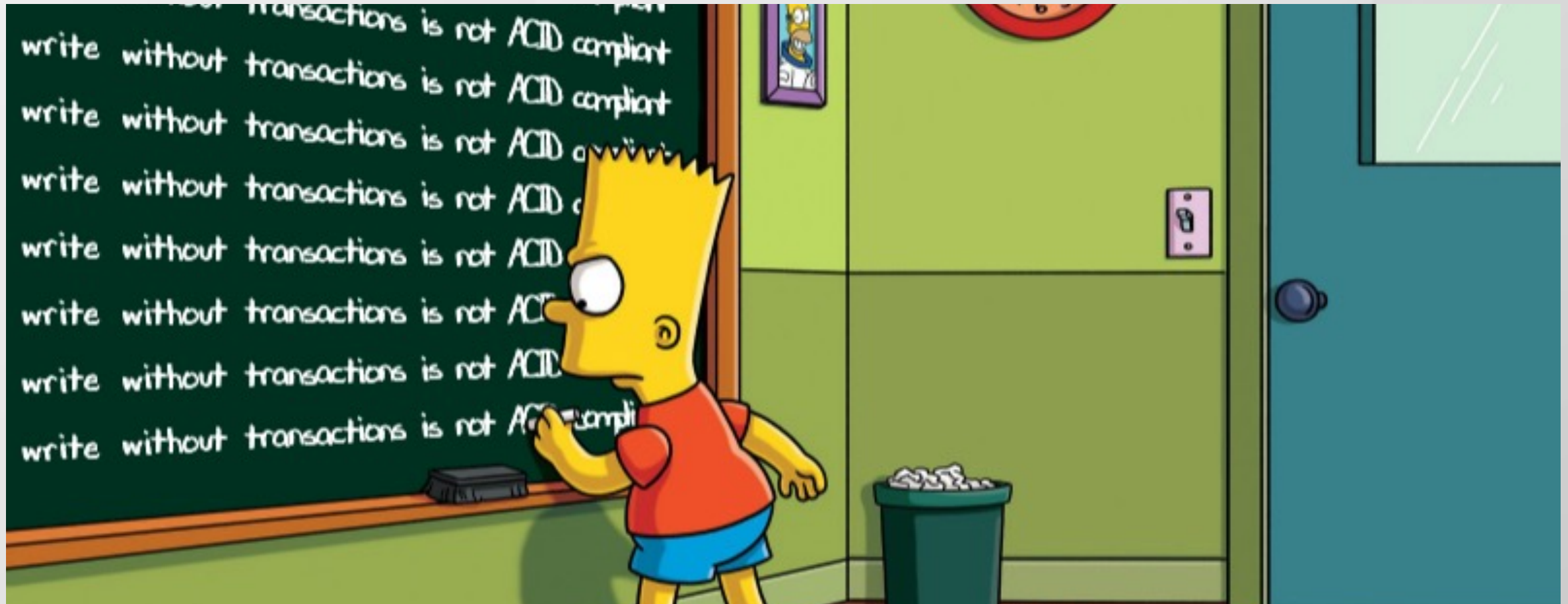
# DEMO 1 - DynamoDB

Write and read items

```
private void insertAndRead(Order order) {  
    // save the item  
    orderTable.putItem(order);  
    LOGGER.debug("saved {}", order);  
  
    // Retrieve the item  
    Order orderRetrieved = orderTable.getItem(order);  
    if (orderRetrieved == null) {  
        insertStats.failedInserted += 1;  
        LOGGER.error("Item retrieved: {} {}, stat: {}%", orderRetrieved, order, insertStats);  
    } else {  
        insertStats.successfulInserted += 1;  
        LOGGER.debug("Item retrieved: {}", orderRetrieved);  
    }  
}
```



# Why did it fail?



## DEMO 2 - DynamoDB

Write with transactions and then read the item

```
private void insertAndRead(Order order) {
    // save the item
    enhancedClient.transactWriteItems(b -> b.addPutItem(orderTable, order));
    LOGGER.debug("saved {}", order);

    // Retrieve the item
    Order orderRetrieved = orderTable.getItem(order);
    if (orderRetrieved == null) {
        insertStats.failedInserted += 1;
        LOGGER.error("Item retrieved: {} {}, stat: {}%", orderRetrieved, order, insertStats);
    } else {
        insertStats.successfulInserted += 1;
        LOGGER.debug("Item retrieved: {}", orderRetrieved);
    }
}
```



# Why did it fail?





## DEMO 3 - DynamoDB

Write with transactions and then read with transactions

```
private void insertAndRead(Order order) {
    // save the item
    enhancedClient.transactWriteItems(b -> b.addPutItem(orderTable, order));
    LOGGER.debug("saved {}", order);

    // Retrieve the item
    List<Document> orders = enhancedClient.transactGetItems(
        b -> b.addItem(orderTable, Key.builder().partitionValue(order.getUuid().toString()).sortValue(order.getUuid().toString()).build());

    if (orders.isEmpty()) {
        insertStats.failedInserted += 1;
        LOGGER.error("Item retrieved: {} {}, stat: {}%", orders.size(), order, insertStats);
    } else {
        insertStats.successfulInserted += 1;
        LOGGER.debug("Item retrieved: {}", orders.get(0));
    }
}
```



# Why did it fail?





## DEMO 4 - DynamoDB

Write with transactions and then read with transactions and handle exceptions

```
private void insertAndRead(Order order) {
    // save the item
    enhancedClient.transactWriteItems(b -> b.addPutItem(orderTable, order));
    LOGGER.debug("saved {}", order);

    // Retrieve the item
    try {
        List<Document> orders = enhancedClient.transactGetItems(b -> b.addItem(orderTable, Key.builder()
            .partitionValue(order.getUuid().toString()).sortValue(order.getCreationTs().toString()).build());

        if (orders.isEmpty()) {
            insertStats.failedInserted += 1;
            LOGGER.error("Item retrieved: {} {}, stat: {}%", orders.size(), order, insertStats);
        } else {
            insertStats.successfulInserted += 1;
            LOGGER.debug("Item retrieved: {}", orders.get(0));
        }
    } catch (TransactionConflictException e) {
        insertStats.failedInserted += 1;
        LOGGER.error("Exception while getting item: {}, stat: {}% {}", order, insertStats, e.getMessage());
        LOGGER.debug("Exception while get item: {}", order, e);
    }
}
```



# Why did it fail?

Transactions might not be isolated

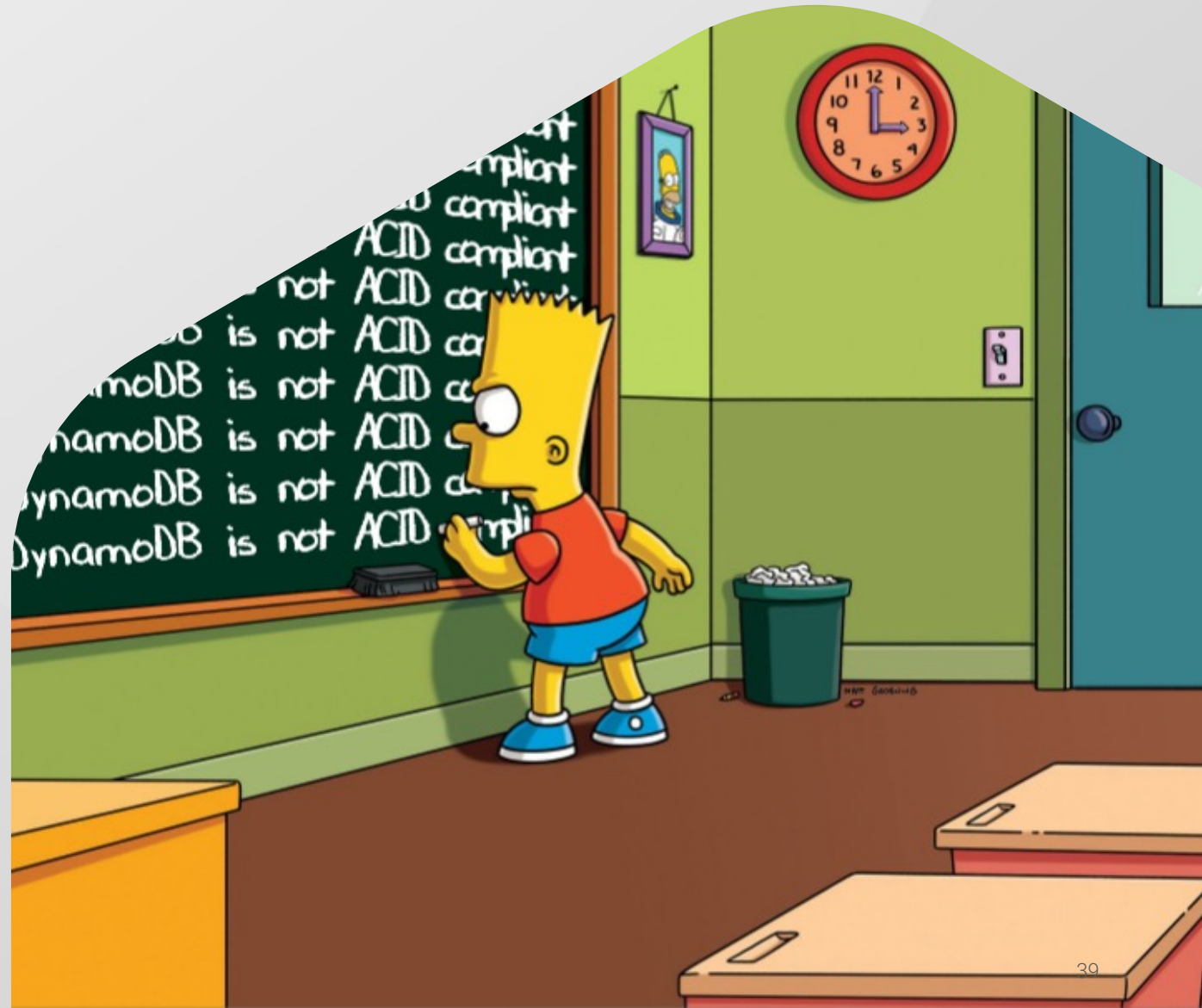
- I is broken

Read consistency not guaranteed

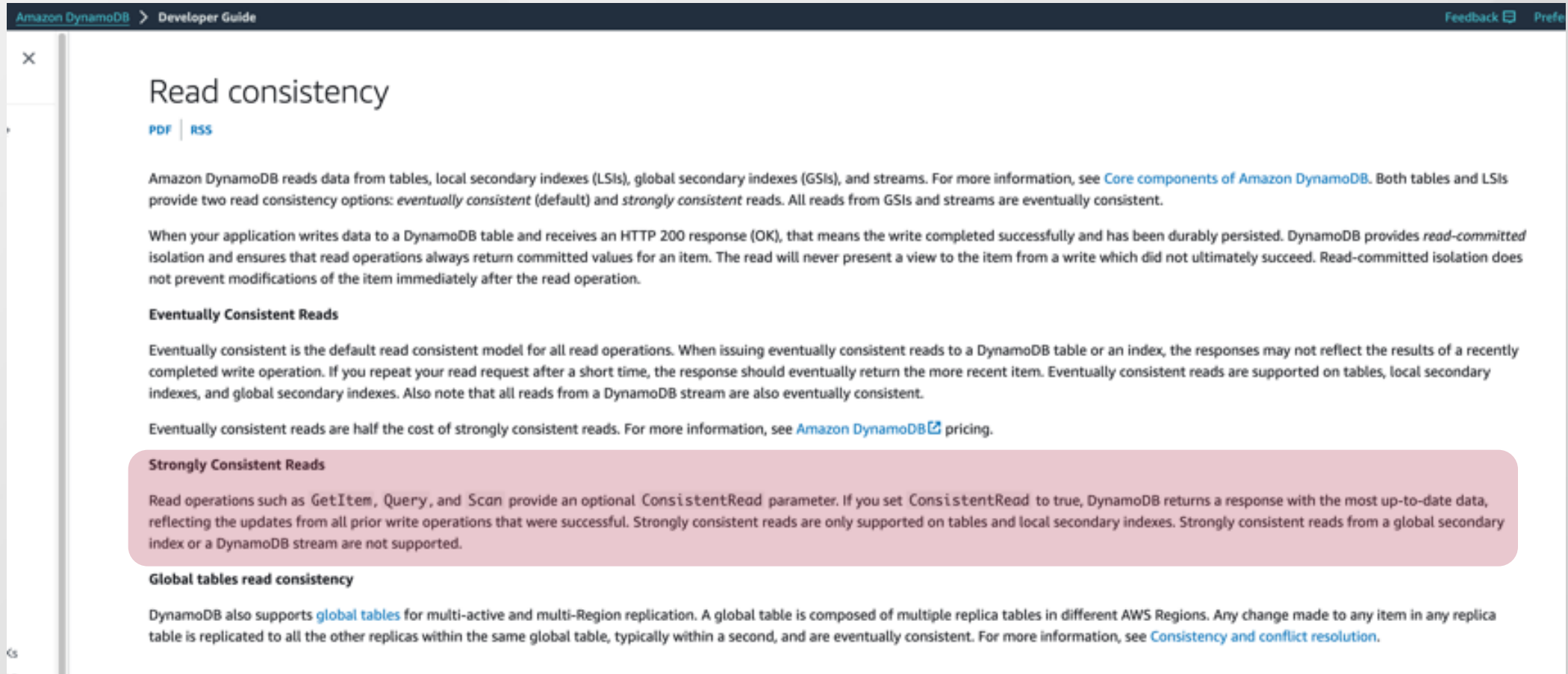
- C is broken

*Eventual consistency and the way AWS implemented Transactions destroys ACID compliance*

*Read and Write cannot be combined within one transaction*



# AWS describes read-consistency as follows



The screenshot shows the Amazon DynamoDB Developer Guide page for 'Read consistency'. The page title is 'Read consistency' with links for 'PDF' and 'RSS'. The main text explains that Amazon DynamoDB reads data from tables, local secondary indexes (LSIs), global secondary indexes (GSIs), and streams. It notes that tables and LSIs offer two consistency options: eventually consistent (default) and strongly consistent. A paragraph describes read-committed isolation, ensuring that reads return committed values. The page is divided into sections: 'Eventually Consistent Reads', 'Strongly Consistent Reads' (highlighted in a pink box), and 'Global tables read consistency'. The 'Strongly Consistent Reads' section mentions that operations like GetItem, Query, and Scan support a ConsistentRead parameter. The 'Global tables read consistency' section explains that global tables consist of multiple replica tables in different AWS Regions, which are replicated and eventually consistent.

Amazon DynamoDB > Developer Guide Feedback Prefe

## Read consistency

[PDF](#) | [RSS](#)

Amazon DynamoDB reads data from tables, local secondary indexes (LSIs), global secondary indexes (GSIs), and streams. For more information, see [Core components of Amazon DynamoDB](#). Both tables and LSIs provide two read consistency options: *eventually consistent* (default) and *strongly consistent* reads. All reads from GSIs and streams are eventually consistent.

When your application writes data to a DynamoDB table and receives an HTTP 200 response (OK), that means the write completed successfully and has been durably persisted. DynamoDB provides *read-committed* isolation and ensures that read operations always return committed values for an item. The read will never present a view to the item from a write which did not ultimately succeed. Read-committed isolation does not prevent modifications of the item immediately after the read operation.

### Eventually Consistent Reads

Eventually consistent is the default read consistent model for all read operations. When issuing eventually consistent reads to a DynamoDB table or an index, the responses may not reflect the results of a recently completed write operation. If you repeat your read request after a short time, the response should eventually return the more recent item. Eventually consistent reads are supported on tables, local secondary indexes, and global secondary indexes. Also note that all reads from a DynamoDB stream are also eventually consistent.

Eventually consistent reads are half the cost of strongly consistent reads. For more information, see [Amazon DynamoDB pricing](#).

### Strongly Consistent Reads

Read operations such as `GetItem`, `Query`, and `Scan` provide an optional `ConsistentRead` parameter. If you set `ConsistentRead` to true, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. Strongly consistent reads are only supported on tables and local secondary indexes. Strongly consistent reads from a global secondary index or a DynamoDB stream are not supported.

### Global tables read consistency

DynamoDB also supports [global tables](#) for multi-active and multi-Region replication. A global table is composed of multiple replica tables in different AWS Regions. Any change made to any item in any replica table is replicated to all the other replicas within the same global table, typically within a second, and are eventually consistent. For more information, see [Consistency and conflict resolution](#).

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html>

# Consistency Model description by AWS

## Consistency Model

[PDF](#) | [RSS](#)

A database consistency model determines the manner and timing in which a successful write or update is reflected in a subsequent read operation of that same value.

Amazon DynamoDB lets you specify the desired consistency characteristics for each read request within an application. You can specify whether a read is eventually consistent or strongly consistent.

The eventual consistency option is the default in Amazon DynamoDB and maximizes the read throughput. However, an eventually consistent read might not always reflect the results of a recently completed write. Consistency across all copies of data is usually reached within a second.

A strongly consistent read in Amazon DynamoDB returns a result that reflects all writes that received a successful response prior to the read. To get a strongly consistent read result, you can specify optional parameters in a request. It takes more resources to process a strongly consistent read than an eventually consistent read. For more information about read consistency, see [Data Read and Consistency Considerations](#).

Apache HBase reads and writes are strongly consistent. This means that all reads and writes to a single row in Apache HBase are atomic. Each concurrent reader and writer can make safe assumptions about the state of a row. Multi-versioning and time stamping in Apache HBase contribute to its strongly consistent model.

<https://docs.aws.amazon.com/whitepapers/latest/comparing-dynamodb-and-hbase-for-nosql/consistency-model.html>



# Rochester Paper about ACID and CAP

## 2.1. ACID Properties

We need to refer the ACID properties[12]:

### **Atomicity**

A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

### **Consistency preservation**

A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

### **Isolation**

A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

### **Durability or permanency**

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

# Rochester Paper about ACID and CAP

## 2.1. ACID Properties

We need to refer the ACID properties[12]:

### Atomicity

A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

### Consistency preservation

A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

### Isolation

A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

### Durability or permanency

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

## 2.2. CAP Theorem

For a distributed database, the CAP theorem states that it's impossible to simultaneously provide more than two out of the following three guarantees:

### Consistency

Every read receives the most recent write or an error

### Availability

Every request receives a (non-error) response – without guarantee that it contains the most recent write

### Partition tolerance

The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Based on CAP theorem, different database picks different combination of consistency, availability, and partition tolerance:

- **CA:** Relational Database
- **CP, AP:** Non-Relational Database

**AWS DynamoDB is ACID-compliant**





## Busted or confirmed?



NoSQL is faster than relational DB



Oracle is the most expensive DB



AWS DynamoDB is ACID-compliant



# Busting some database-myths

episode 1

Thanks for joining

MYTH

FACT

Erich Steiger

LinkedIn

Follow @ErichPSteiger

<https://erichsteiger.com/>

## Upcoming Talks

18<sup>th</sup> June 2024: DB Roundtable

Select AI chat what is TWINT?



TWINT