Database Performance, Security, and NoSQL

School of Information Studies
Syracuse University

PERFORMANCE FACTORS

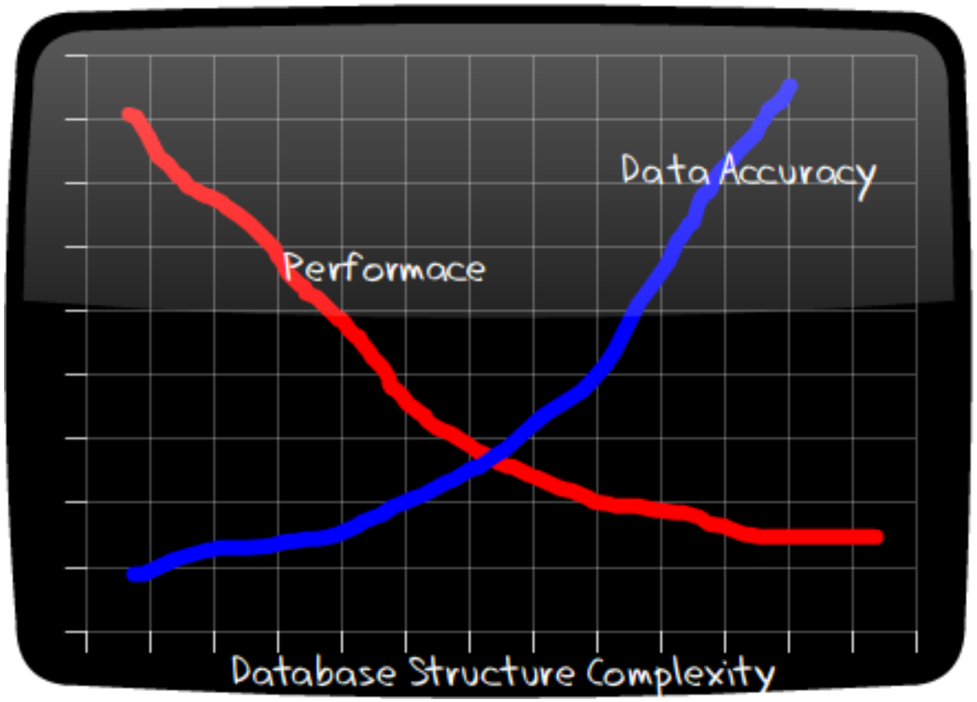
What constitutes poor RDBMS performance, and which factors lead to it?

Factors That Impact Physical Design

Structure	Table and relationship count
Volume	Quantity of data
Volatility	Rate of change of data
Input Mode	How does data get in
Storage Format	Data type selection
Retrieval	Optimizing data retrieval

Structure

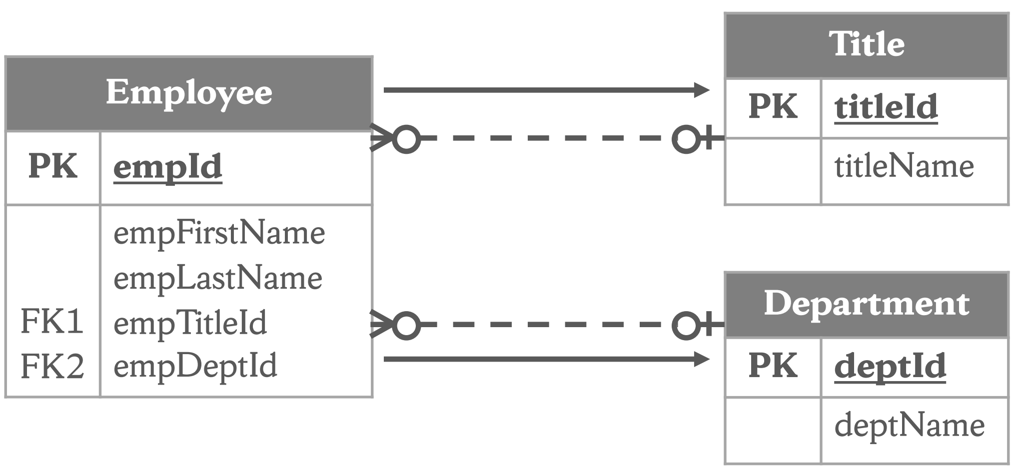
Structure is the trade-off between performance and data accuracy.

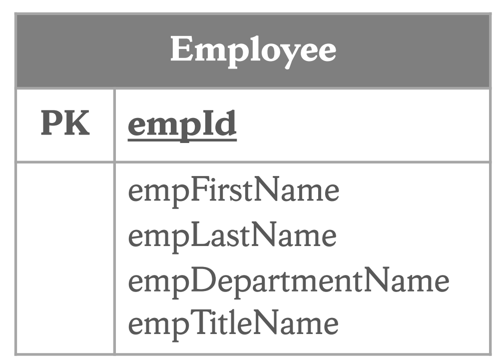


School of Information Studies
Syracuse University

Example: Structure

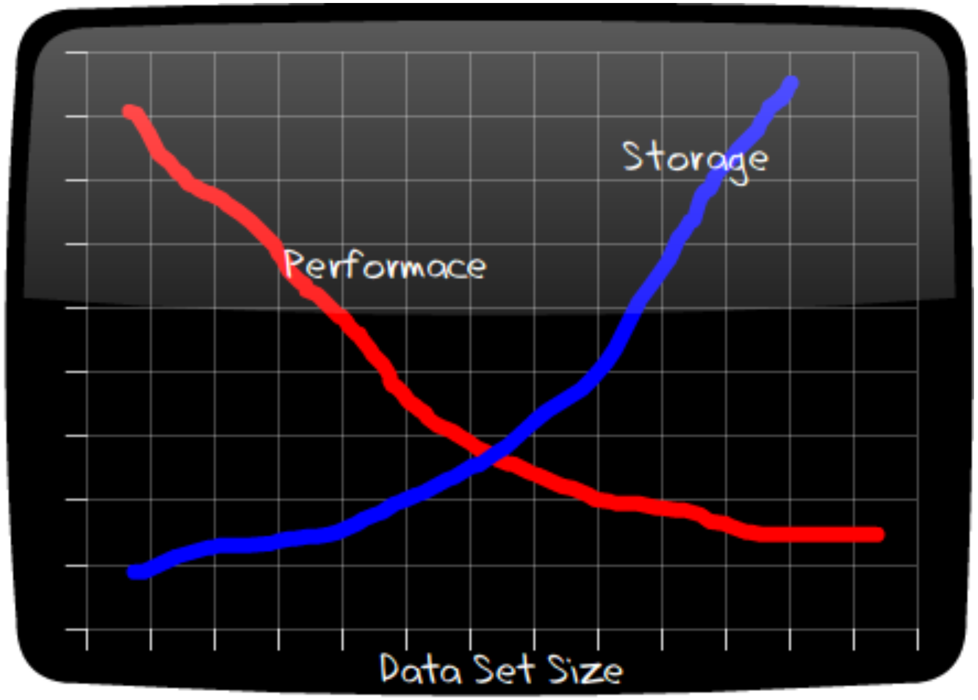
Complex Structure	Simplified Structure
More normalized	Less normalized
Less chance of bad data	Greater chance of bad data
Will perform poorly at large scale	Better performing at scale





Data Volume

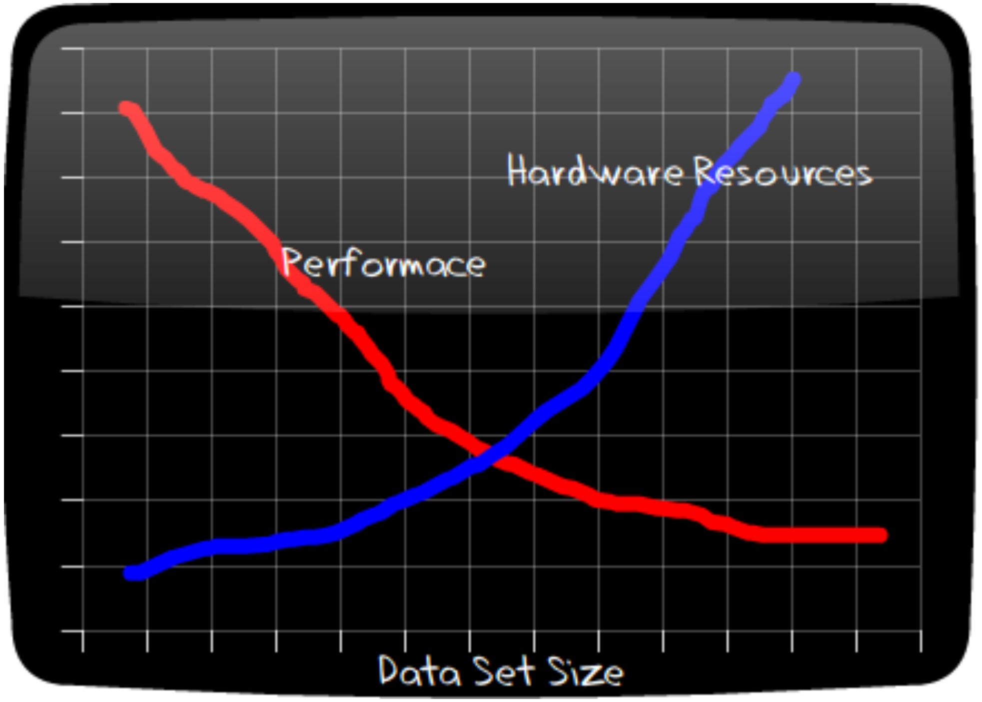
How much data do you have? Volume determines your hardware and DBMS selection. Can you scale horizontally?



School of Information Studies
Syracuse University

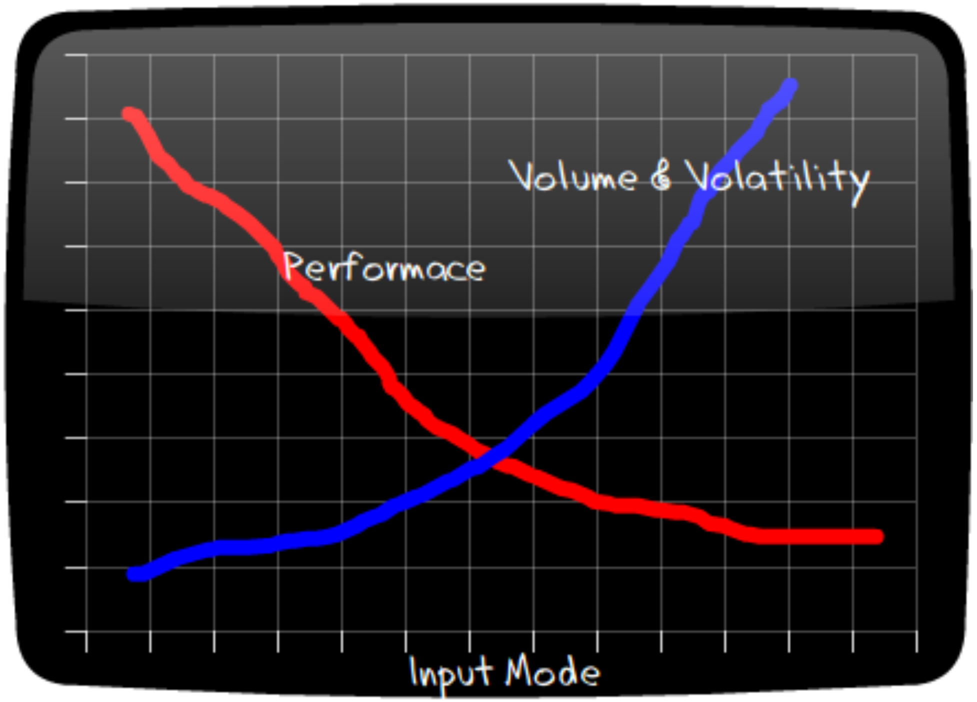
Volatility

The more frequently your data change, the more hardware resources should be dedicated to dealing with changes in data.



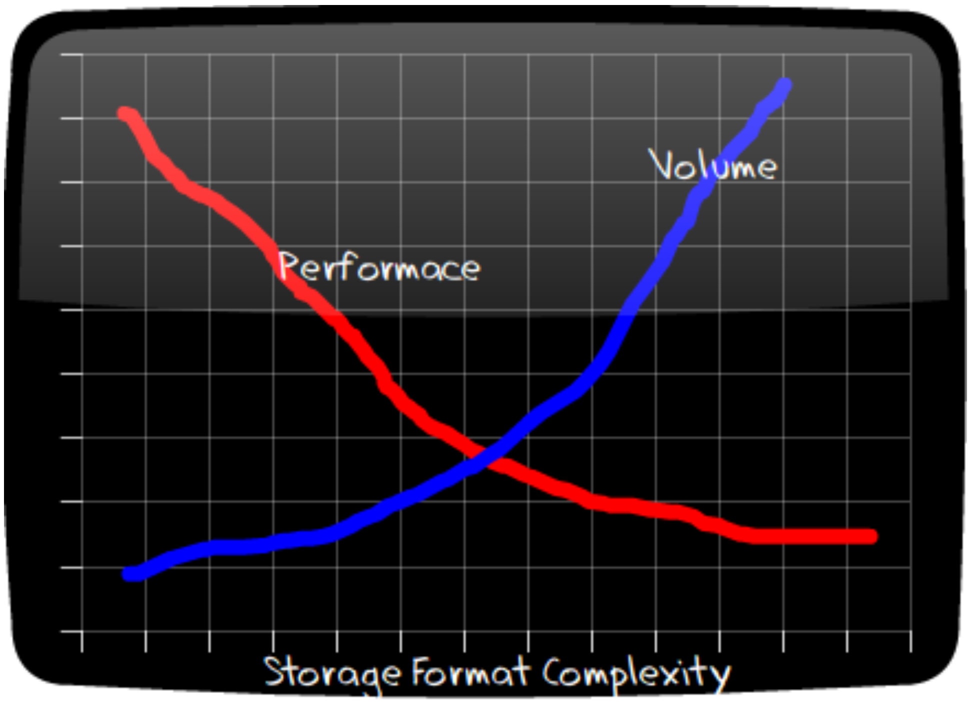
Input Mode

How are the data getting in? High input modes need simplified table designs and more hardware resources to combat volume/volatility.



Storage Format

How you choose to store your data (your table designs) will effect volume and performance.



School of Information Studies
Syracuse University

Storage Format

Right-size your column data types. Using more than you need negatively affects volume and performance.

Data Type	Range	Storage
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 bytes
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 bytes
tinyint	0 to 255	1 byte

For SQL Server: http://msdn.microsoft.com/en-us/library/ms187752.aspx

Storage Format

Data Type Decisions

`char` or `varchar`?

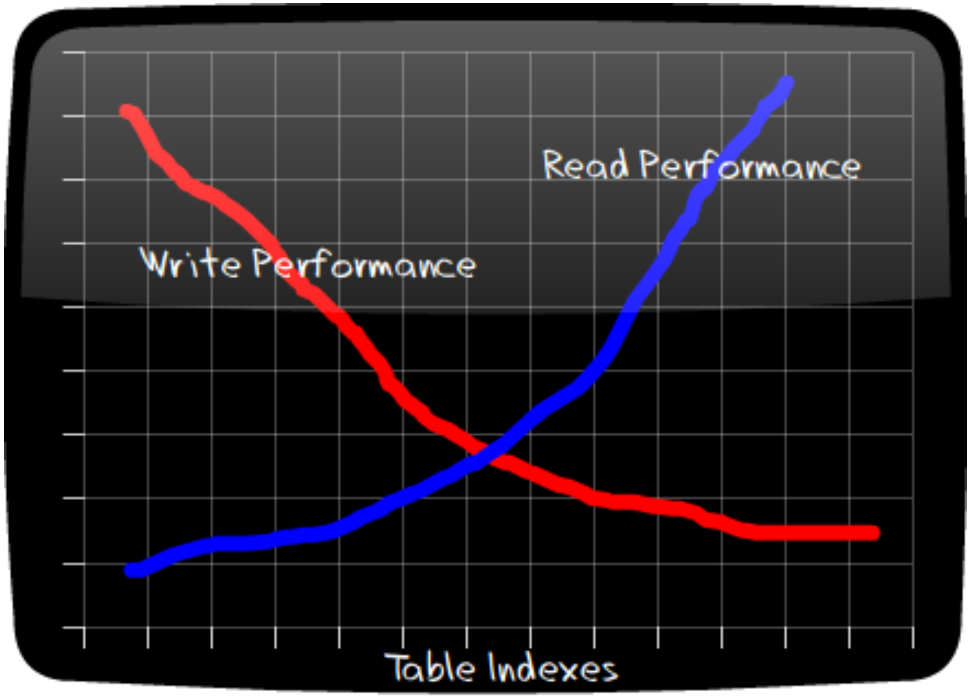
- `char` faster than `varchar`
- `char` uses more space than `varchar`

`int` or `decimal`? or `float`?

- Always choose `int` unless you need `decimal`
- Same for `decimal` vs. `float`
- Never use `char` for true/false—use `bit`.

Retrieval

Optimizing data retrieval is the easiest factor you can improve; it impacts write operations adversely.



Retrieval

Index: a thread over columns in a table that speed up searches over it

Cluster: physically grouping rows of data in the same physical block on disk; only one clustered index per table

Products		
ID	Name	Department
1	Hammer	Hardware
2	T-shirt	Clothing
3	Wrench	Hardware
4	Socks	Clothing
5	TV set	Electronics
6	Shoes	Clothing
7	Drill	Hardware

Department Index	
Department	IDs
Clothing	2,4,6
Electronics	5
Hardware	1,3,7

As data change, the index must be rebuilt, hence the negative impact on write performance.

PHYSICAL MODEL OF SQL SERVER

Physical Database Design

Where are your data stored? Does it matter?

It does if you care about performance!

The Physical Abstraction in SQL Server

Server: installed on hardware, licensed by hardware

Instance: one or more “setups” of SQL Server on the same physical server (Test, prod, dev.)

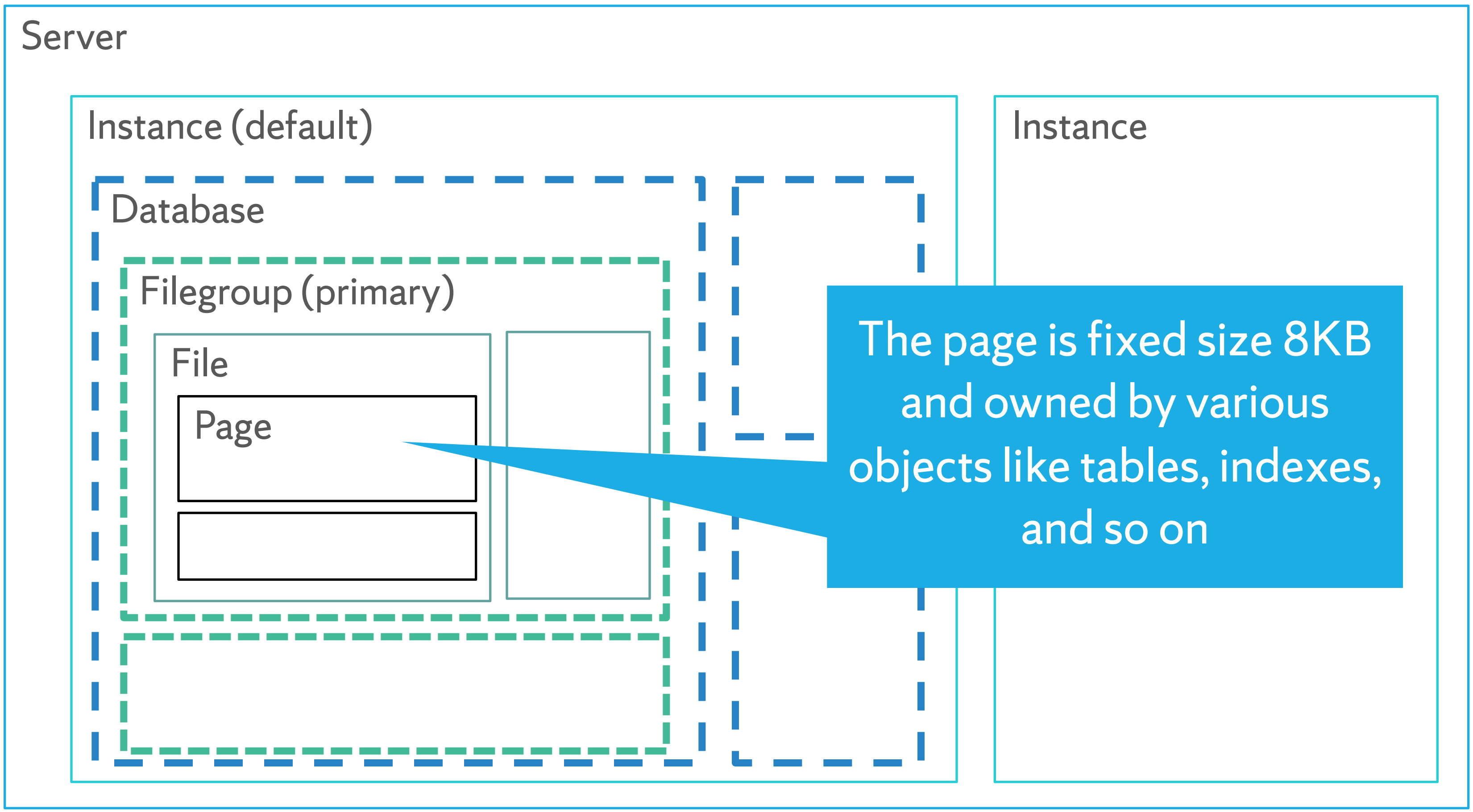
Database: stored as one or more files

- *databasenname*.mdf → primary file, member of the PRIMARY filegroup
- Othername.ndf → secondary files used to spread data across physical partitions/disks
- *databbasename*_log.ldf → transaction log file holds transactions/recovery information

Filegroup (tablespace): a logical name for one or more physical files

Page: set of continuous table rows inside a physical file

SQL Server Physical Model



Physical Performance Tricks

If your server has more than one disk, create one file per disk in the same filegroup to spread IO over the disks.

If your server has faster SSD (solid-state disks), create a filegroup on that disk to support high-velocity tables.

Do not store the database files on the same disk/partition as the operating system and SQL server itself. Minimize your IO contention.

CLUSTERED INDEXES

Database Index

Improves the searchability of our data, at the expense of maintaining multiple versions of it.

Just like the index in the back of a book takes us to a page where that word occurs…

The database index takes us to the database page where the data for which we're searching can be found.

Scan traverses linearly, a page at a time.

Seek jumps to the page with the content.

Clustered Index

Sorts and stores the data in the order of the key values.

There can only be one clustered index per table since rows can only be stored in that order.

By default, the primary key contains a clustered index, but this is not a requirement.

The pages are organized by the clustered index.

Primary Key as a Clustered Index

This is why auto-incrementing values are good for clustered indexes—it avoids page fragmentation

- Int identity
- Date/time stamp

Ideal key/clustered index

- Narrow—not many bytes in size
- Unique
- Static—never changing
- Ever-increasing

Necessary if you use JOIN, ORDER BY, or WHERE

Demo: Create Table Revisited

Create table clustered/non-clustered

You can store the index in the same filegroup or a different filegroup from the table

Create a filegroup through GUI, generate script

Add table to filegroup

NON-CLUSTERED INDEXES

Non-Clustered Index

Secondary indexes compared with the primary clustered index

Use to improve the performance of queries

You can create multiple non-clustered indexes on your table

The cost is extra space, and extra inserts

Every index on a table must be updated when the data in a table are updated

What to index?

- Columns in WHERE, GROUP BY clauses
- Columns with many distinct values

CREATE INDEX

```
CREATE [UNIQUE] INDEX index_name
     ON table_name (column,[…])
     [INCLUDE (column, […])]
```

The columns next to the table are the index keys.

The index can be made unique—a constraint.

The columns in the INCLUDE clause are not part of the index key.

When these columns are in the projection of the SELECT statement, you get an index seek.

Scans and Seeks: Finding a Row of Data

Worst

Table scan: search for—row by row—looking for data in each column; worst-case scenario

Clustered index scan: use the clustered index (primary key, usually) to search row by row

Clustered index seek: jump to the correct page using the clustered index

Index scan: search the index row by row—better than scanning the clustered index!

Index seek: uses the index to jump to the correct page—Nirvana!

Best

Demo: Scans vs. Seeks

Execution plans, so you can see how the table was read

Let's explore indexing, scans, and seeks

INDEXING RULES OF THUMB

Non-Clustered Index Rules of Thumb

Use Indexes

The table contains over 100,000 rows

The searchable field (indexed column) has a wide range of values

The searchable field has a large number of null values

The searchable field is queried frequently

Queries retrieve less than 2-4% of the table's rows

Avoid Indexes

There are relatively few rows in the table

The column is not used for searching

The majority of the queries that retrieve more than 2-4% of the table's rows

There is high insert or update transaction volatility

Rebuilding Indexes

As data are inserted, updated, and deleted into your tables, your indexes will no longer be optimal

The pages will not be stored efficiently, and your indexes become fragmented

Reorganize: move the pages around

Rebuild: create the index from scratch

```
ALTER INDEX index_name ON table REORGANIZE
ALTER INDEX index_name ON table REBUILD
```

Demo: Index Statistics

Reports: index usage statistics

Reports: index physical statistics

REBUILD INDEX

COLUMNSTORE INDEXES

Columnstore Index

The standard for storing and querying large tables with many columns

Rather than storing the data in rows, the data are stored in columns

Appropriate for tables with many columns (wide tables) and those with many similar values as found in data warehouse fact tables

Comes in clustered/non-clustered versions

Data are compressed and cached in memory

Tangent: Columnstore?

Physical Row Store

Row 1	US
	Alpha
	3,000
Row 2	US
	Beta
	1,250
Row 3	JP
	Alpha
	700
Row 4	UK
	Alpha
	450

Easier to retrieve a column

Easier to aggregate values or distinct values in a column

Used in analytics

Logical Table

Country	Product	Sales
US	Alpha	3,000
US	Beta	1,250
JP	Alpha	700
UK	Alpha	450

Easier to add new rows

Simple to retrieve a row, update a row

Used for CRUD operations

Physical Col Store

Country	US
	US
	JP
	UK
Product	Alpha
	Beta
	Alpha
	Alpha
Sales	3,000
	1,250
	700
	450

CREATE COLUMNSTORE

```
CREATE [CLUSTERED|NONCLUSTERED]
     COLUMNSTORE INDEX index_name
     ON table_name (column,[…])
```

Can be clustered or non-clustered; only one clustered index

The columns specified are not keys, they are the columns included in the column store index; the keys are the names of the columns!

Demo: Columnstore Index

Run query on orders table and look at live statistics

Make a non-clustered index to improve this query

But it doesn't work for other queries—we'd need another index

Columnstore to the rescue

INDEXED VIEWS

Indexed Views

Easy way to improve the performance of a view

An indexed view is SQL Server's version of a materialized view

The output of the view is saved into an index table to improve read performance of the view

When the underlying data are updated, so is the view

Must bind the schema of the underlying tables—cannot alter the underlying table schema without dropping the view first

Rules for Indexing a View

Use WITH SCHEMABINDING in the CREATE VIEW statement

The view must be deterministic—the same input yields the same output

All columns must be specified; no wildcard columns

Tables must be schema qualified

Must be a unique clustered index (like setting a PK)

Demo: Indexed Views

Start with v_products_with_vendors view

Cannot index a view

Add with schemabinding

Fix columns

Add two part schema

Security Concepts

Database Security

Protection of the data against accidental or intentional loss, destruction, or misuse

Increased difficulty due to Internet access and client-server technologies

More exposure = greater risk!

Key Concepts

Principal: an entity that is given permission to a something; typically a user but can be a resource or group

Securable: the something to which the principal is given access; typically a database object such as a database, table, stored procedure, view, and so on

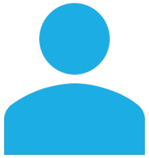
Permissions: the rights the principal has to the securable

Authentication vs. Authorization

Authentication

Verifying a principal

Login



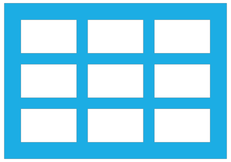
Yes, that's Tom in Accounting

Authorization

Verifying a principal's permissions

Done when access is requested

Tom would like to READ the payroll table



Hashing and Encryption

Hashing

A unique number is generated from the input

This is a one-way process and cannot be reversed

Used with passwords or to generate unique values

MD5, SHA1, SHA2

Encryption

A unique output is generated from the input

The process is reversible

Requires keys to encrypt and decrypt

Used whenever the value must be unencrypted

AES, PGP

Ask Yourself

What are you securing?

- Access to the data?
- Or the data itself?

Whom are you securing against?

- Authenticated Principals
- Unauthenticated Principals

Best Practices

- Start with no permissions
- Only add what its necessary

Security Authorization Matrix

Principal	Securable	Permissions	Constraints
Accounting Dept.	Customer record	READ, INSERT	None
Tom	Customer record	INSERT, UPDATE	Credit limit <1000
Kiosk 145	Customer record	UPDATE	Self only
Website	Customer record	READ	Names only

No need to enumerate default access—it should be none!

Security Manipulation Language

```
CREATE LOGIN loginname
WITH PASSWORD=N'pass'
      ,DEFAULT_DATABASE=dbname
      ,CHECK_EXPIRATION=OFF
      ,CHECK_POLICY=OFF
```

```
CREATE USER username from login loginname
```

```
GRANT permission ON object TO principal
```

```
REVOKE permission ON object TO principal
```

```
DENY permission ON object TO principal
```

Demo: Creating Accounts

Make a login

Make an account

Search for the principal in sys.database_principals

Log in with the account in another tab

Account had no rights to anything

DEMO: PERMISSIONS AND SECURABLES

Permissions and Securables

Permissions

INSERT
UPDATE
DELETE
SELECT
EXECUTE

Securables

DATABASE
SCHEMA
OBJECT
TABLE
COLUMN
VIEW
PROCEDURE
FUNCTION

Demo: GRANT, DENY, and REVOKE

Let's demonstrate how security works through examples using the account table.

Why Add noSQL Features to Relational Databases?

Convergence!

Newer versions of SQL Server have added the following NoSQL features:

- JSON and XML support for document databases… like MongoDB
- Query support for external data… like Apache Drill
- In-memory database tables… like Redis
- File tables where a folder on disk is a table… like Hadoop HDFS
- Graph database support… like Neo4J
- Columnstore support… like HBase and Cassandra
- Runs on Linux… like most open source software!

It's still a CA system that doesn't scale, but if your data aren't "big data" it's a solid choice. Best of both worlds!

Document Database Features of SQL Server

JSON: JavaScript Object Notation support

XML: Extensible Markup Language support

Transform those formats into tables and back to XML/JSON

Query into those formats

Update those formats

Why Use the Document Features?

Makes sense for data that "stay together" like comments to a blog, or the customer's view of an order

Most APIs use JSON format—why write that in Java or Python?

If the data are going to end up on a website, or in a mobile app, they will probably end up in JSON format!

Demo: JSON

JavaScript Object Notation

A simple, organized, easy data format that is easy for both people and computers to read

Commonly used in APIs and Web applications

Demo: load sample.json


```json
[
    {
        "id": 1,
        "name": "Bike-Pump",
        "price": 15.0000,
        "reviews": [
            {
                "Reviewer": {
                    "Name": "Erin Detyers",
                    "Email": "edt@mail.com",
                    "Rating": 5
                }
            }
        ]
    },
    {
        "id": 2,
        "name": "Handlebars",
        "price": 30.0000,
        "reviews": [
            {
                "Reviewer": {
                    "Name": "Kent Belevit",
                    "Rating": 2
                }
            },
            {
                "Reviewer": {
                    "Name": "Artie Choke",
                    "Email": "ack@mail.com",
                    "Rating": 3
                }
            }
        ]
    },
    {
        "id": 3,
        "name": "Seat",
        "price": 40.0000
    }
]
```

JSON SUPPORT IN SQL SERVER

JSON Support in SQL Server

ISJSON—tests whether a string contains valid JSON

1==yes, ==no

JSON_MODIFY()—changes the value of a JSON string

JSON_QUERY()—extracts an object from a JSON string

JSON_VALUE()—extracts a scalar value from a JSON string

SELECT … FOR JSON AUTO—returns a table in JSON format

OPENJSON()—a table function to return a table from the JSON

Tangent: APPLY Join Operator

APPLY is a special SQL join that, for every row in the left, applies the table function on the right.

The usual format is:

```
FROM X CROSS APPLY Y(X.column)
```

This applies table function **Y()** to each value of **X.column**.

There's an "outer" version that does not filter out nulls in X.column:

```
FROM X OUTER APPLY Y(X.column)
```

Demo: Products Table

Create our table with a JSON field for reviews

Insert data

Insert invalid JSON—fails check constraint

Show the data

Demo: JSON Querying

Use the table function CROSS APPLY OPENJSON() to transform JSON data back into a table-like format.

Use the SQL SELECT statement to transform any query into JSON.

Use JSON_QUERY() to handle issues strings of JSON.

JSON to TABLE to JSON again!