Advanced Database Design Patterns in Action

Patterns

Primary Key

Primary Key – Section Overview

01 Key Types

02 Primary Key Properties

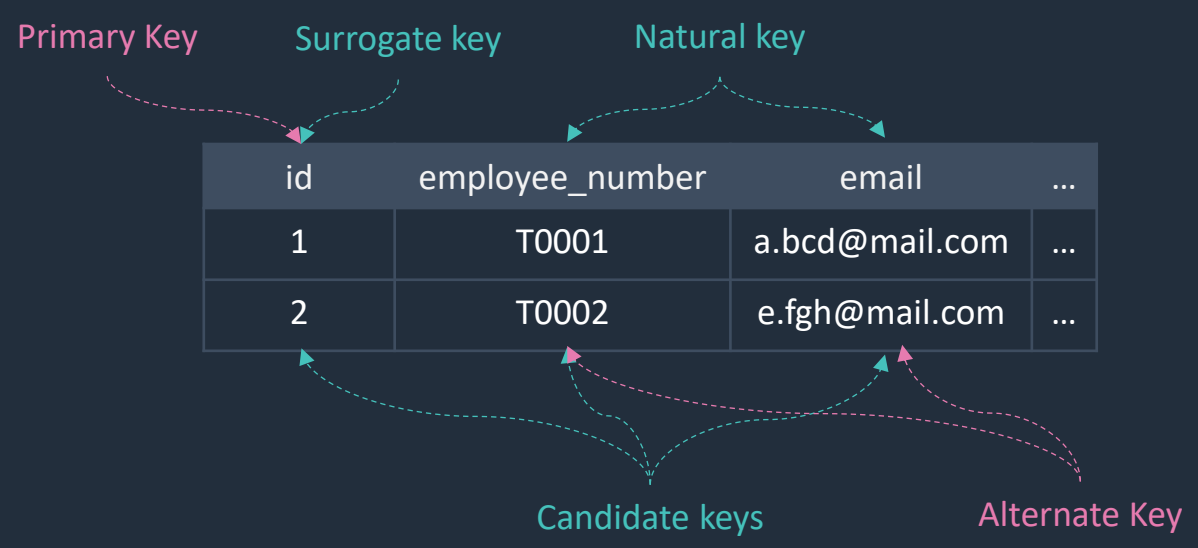
03 Surrogate Key

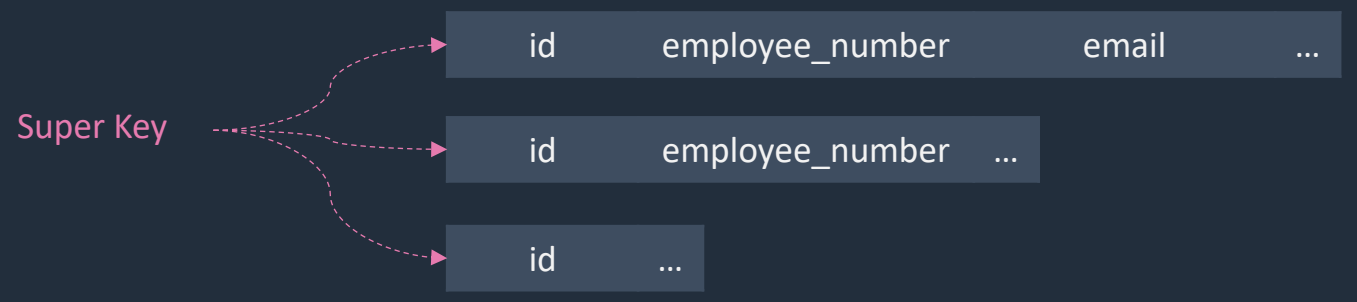
04 Natural Key

05 Hands-On

Key Types

- Super key
- Candidate key
- Primary key
- Alternate key
- Natural key
- Surrogate key





Primary Key Properties

- Uniqueness
- Non-nullability
- Immutability
- Minimality

Surrogate Key

Surrogate Key Pros

- Less likely to change
- Often are shorter than natural keys
- If integers are used, better performance when joining tables and filtering
- No confidentiality constraints

surrogate key example

product_id	product_id
1	PRD00345
2	PRD00617
3	PRD00321

natural key example

product_id
PRD00345
PRD00617
PRD00321

Surrogate Key Pros (continued)

- Easier to write application code, database code or use code generators

product

product_id	...
1	...
2	...

promotion

promotion_id	...
3123	...
3124	...

order

order_id	...
1001	...
1002	...

vendor

vendor_id	...
56	...
57	...

Surrogate Key Pros (continued)

- Simpler primary keys and referencing them foreign keys
- Simpler conditions on joins

surrogate key example

promotion

promotion_id	product_id	effective_start_date	effective_end_date	discount_percent
1	PRD0047	2024-01-01	2024-02-01	30
2	PRD0047	2024-01-01	2024-03-01	35

order_item

order_id	item_id	product_id	promotion_id
1	1	PRD0047	1
2	1	PRD0047	2

natural key example

promotion

product_id	effective_start_date	effective_end_date	discount_percent
PRD0047	2024-01-01	2024-02-01	30
PRD0047	2024-01-01	2024-03-01	35

order_item

order_id	item_id	product_id	effective_start_date	effective_end_date
1	1	PRD0047	2024-01-01	2024-02-01
2	1	PRD0047	2024-01-01	2024-03-01

Natural Key

Natural Key Pros

- Use data that is already present in the record
- Use data that is recognizable by humans and has a business meaning
- Possibly fewer joins to collect necessary data

natural key example

product_number
PRD00345
PRD00617
PRD00321

order_item

product_number	order_item
PRD00345	23
PRD00617	24
PRD00321	25

product

product_number
PRD00345
PRD00617
PRD00321

surrogate key example

product_id	product_number
1	PRD00345
2	PRD00617
3	PRD00321

order_item

product_id	order_item
1	23
2	24
3	25

product

product_id	product_number
1	PRD00345
2	PRD00617
3	PRD00321

Natural Key Pros (continued)

- Natural relationships are not hidden
- Require less constraints

natural key example

PK

country_name	...
BRAZIL	...
INDIA	...
UNITED_STATES	...

country_name	state_name	...
BRAZIL	BAHIA	...
INDIA	KARNATAKA	...
UNITED_STATES	NEW_YORK	...

state_name	city_name	...
BAHIA	SALVADOR	...
KARNATAKA	BANGALORE	...
NEW_YORK	NEW_YORK	...

surrogate key example

PK UNIQUE

country_id	country_name	...
23	BRAZIL	...
67	INDIA	...
123	UNITED_STATES	...

country_id	state_id	state_name	...
23	1	BAHIA	...
67	2	KARNATAKA	...
123	3	NEW_YORK	...

state_id	city_id	city_name	...
1	1	SALVADOR	...
2	2	BANGALORE	...
3	3	NEW_YORK	...

Surrogate Kye vs Natural Key - Summary

- Trade-offs
- Requirements

Indexing Strategy

Indexing Strategy – Section Overview

01 Heap

02 Index Basics

03 Nonclustered Index

04 Clustered Index

05 Nonclustered Primary Keys

06 Data Access Operators

07 Query Patterns

08 Join Algorithms

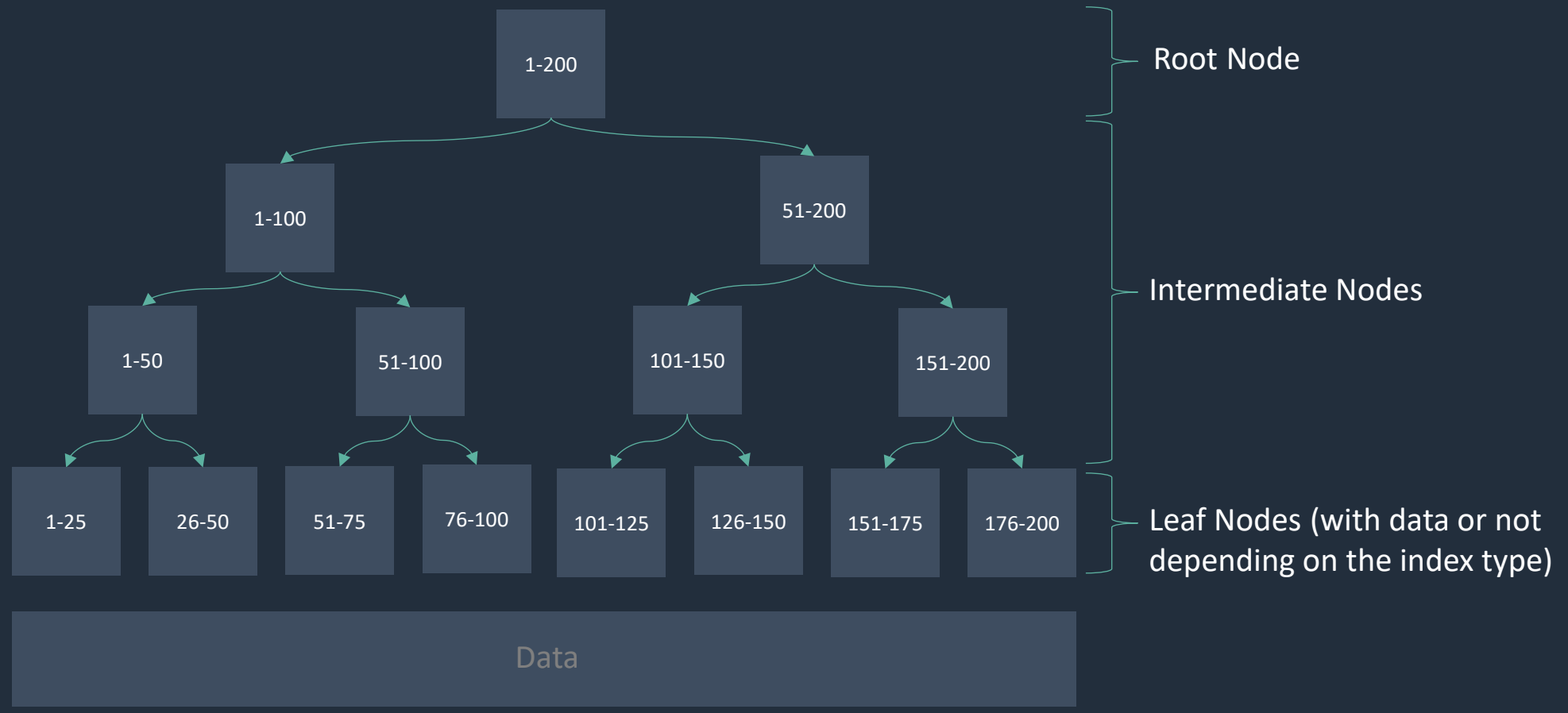
09 Hands-On

Heap

- A heap is a table structure that does not have a clustered index
- There is no key order
- Row identifier FileID : PageID : SlotID
- Faster inserts than in clustered tables
- Suitable when ordered data access is not paramount
  - Staging tables
  - Logging tables
  - Data is always accessed through nonclustered indexes
- Subject to fragmentation

Index Basics

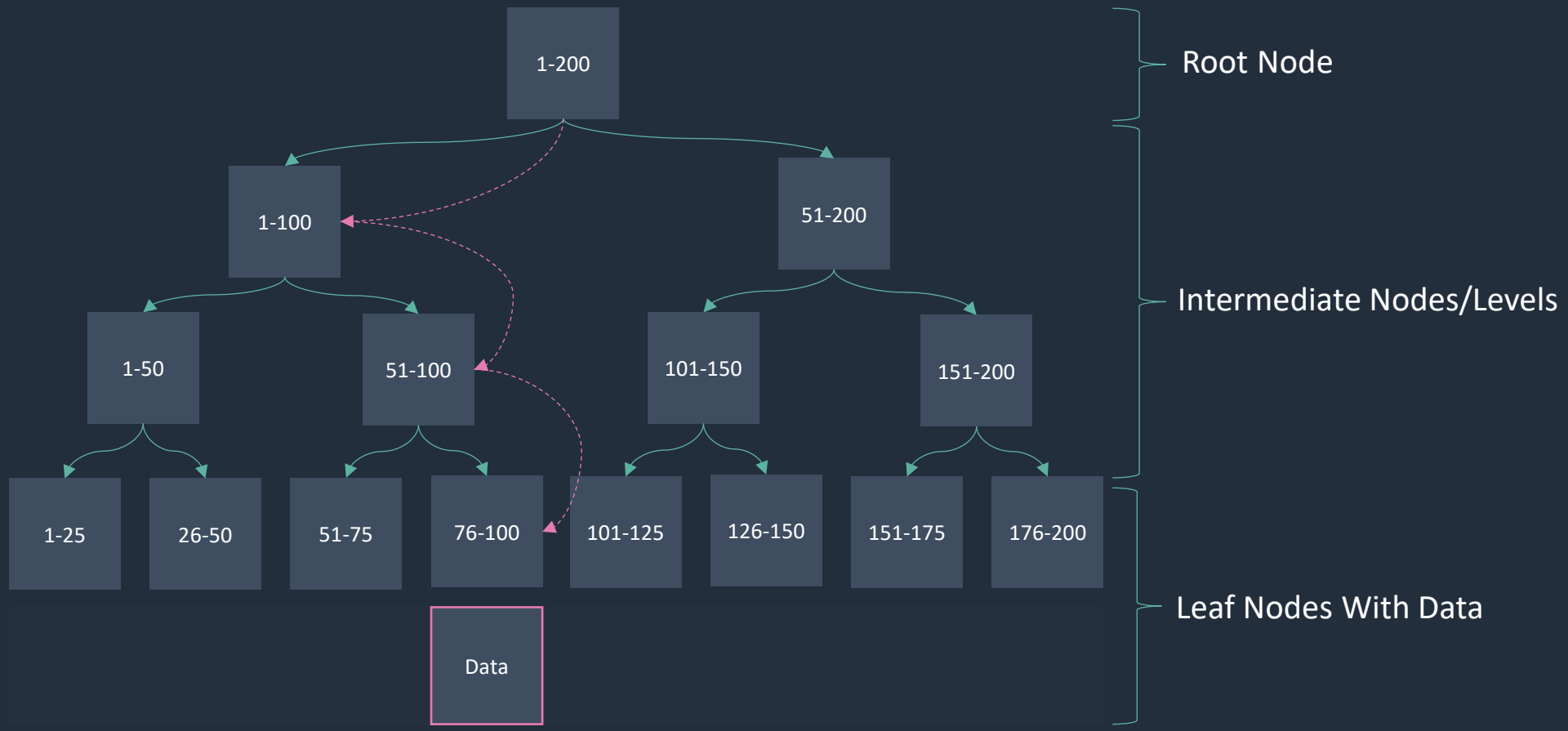
- An index is a data structure that improves the speed of data retrieval operations on a database table
- Implemented as a B-tree
- Can be created on a one or more columns
- Stored on pages
- Carries overhead



Index Traversal Example

```
SELECT *
FROM customer
WHERE customer_id = 77
```

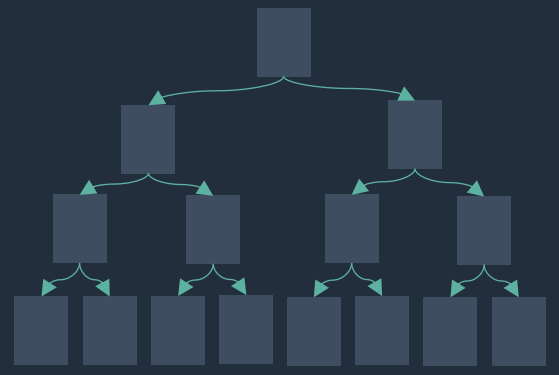
➡ *Data*



Nonclustered Index

- Nonclustered indexes are secondary indexes
- Stored separately
- Multiple indexes can be created
- Improve query performance
- Incur overhead during DML
- Limited to 1700 bytes bytes per key

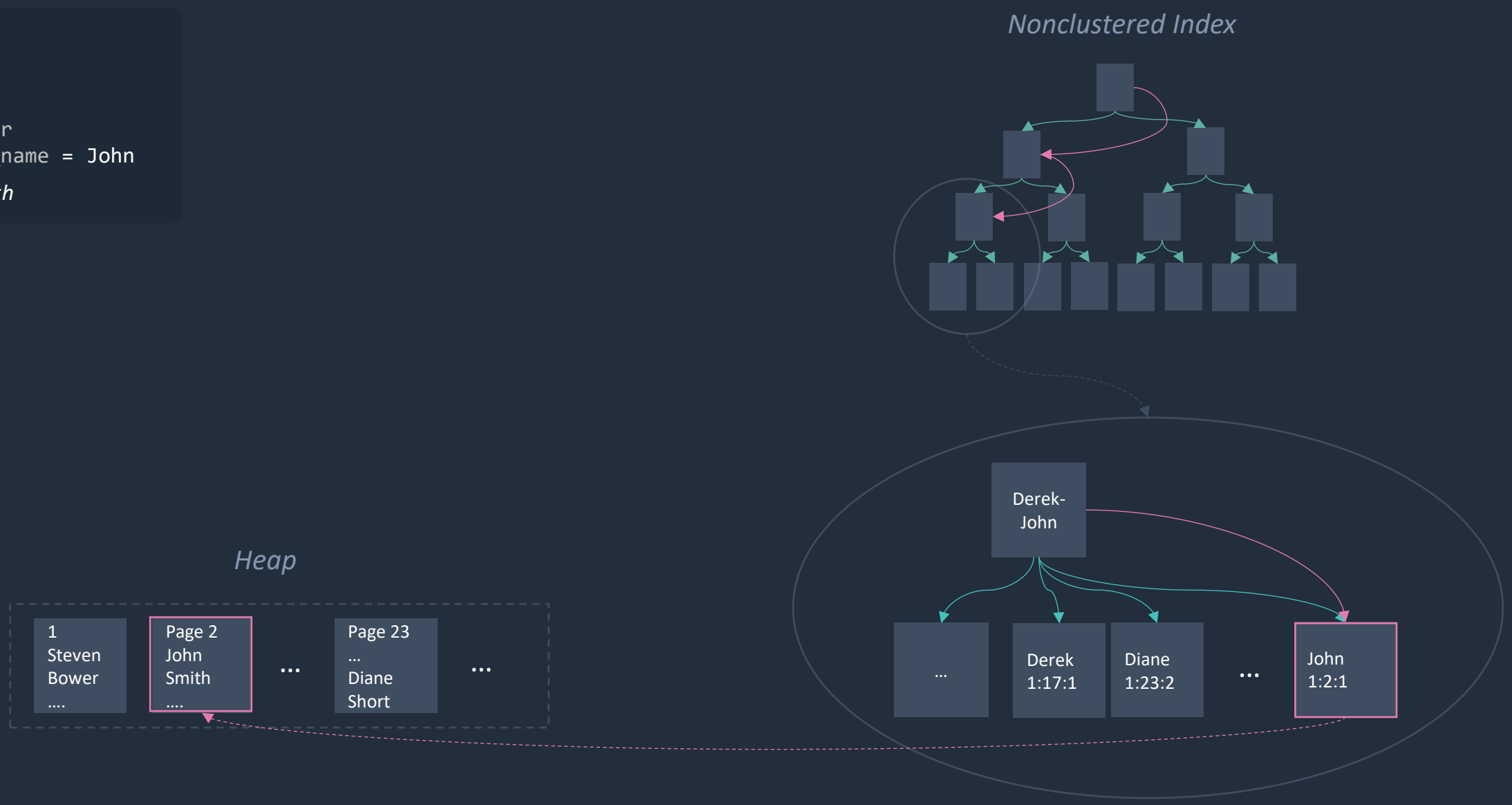
Nonclustered Index B-Tree



Nonclustered Indexes On A Heap

```
SELECT
 first_name,
 last_name
FROM customer
WHERE first_name = John
```

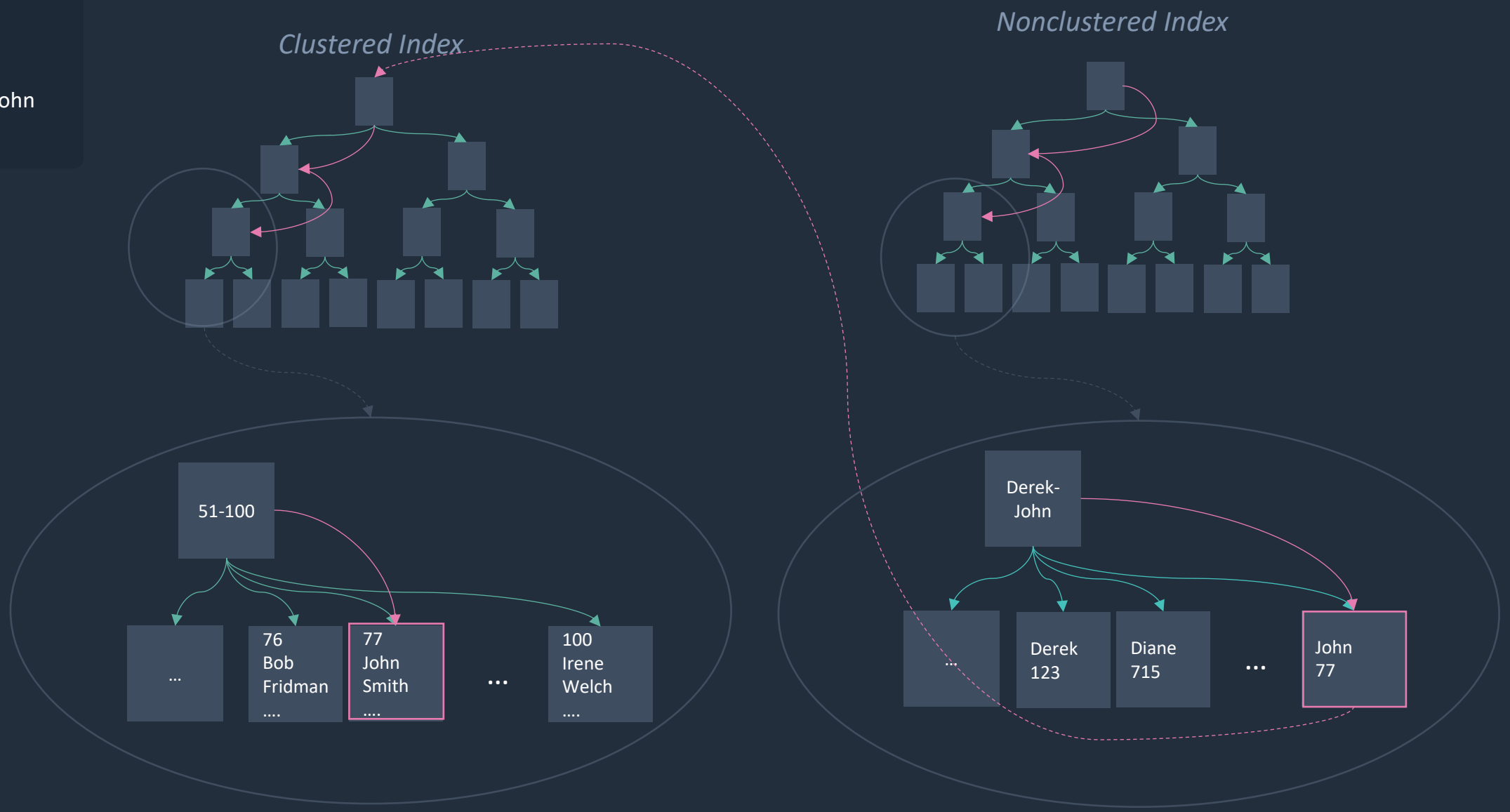
➡ *John Smith*



Nonclustered Indexes On A Clustered Table

```
SELECT
 first_name,
 last_name
FROM customer
WHERE first_name = John
```

➡ *John Smith*



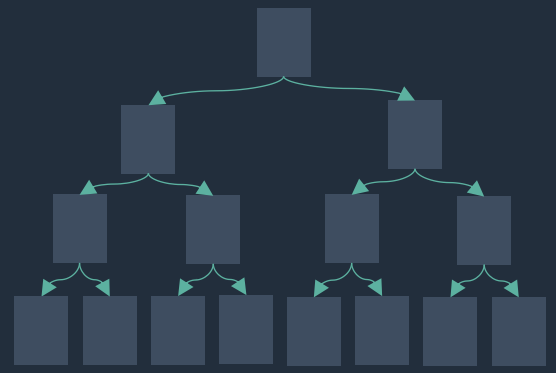
Nonclustered Index Types And Use Cases

- Unique
  - Queries that are not covered by the clustered index
- With included columns
  - Covering index
- Filtered
  - Queries that target a specific small subset of rows
  - Queries that target tables with large amounts of sparse data
  - Queries that target tables with active and archived data
- Computed column
  - Queries that filter and sort based on the computed column

Clustered Index (Clustered Table) - Introduction

- Organizes data within a table based on the index key values
- Only one per table
- Often created on a primary key column
- Page splits cause index fragmentation
- Require maintenance

Clustered Index B-Tree



Clustered Index Design

A clustered index's main function is to provide efficient retrieval of data for critical queries

Key properties

- Unique
- Narrow
- Static
- Increasing

Typically used for queries that:

- Use range operators such as >, >=, <, <= and BETWEEN
- Large data sets retrieval
- Use JOIN clauses
- Use ORDER BY or GROUP BY clauses

Clustered Index On An Integer Column

- IDENTITY vs manually managed

Pros

- Compact, 4 bytes (INT) or 8 bytes (BIGINT)
- For an ever-increasing column less index maintenance
- Smaller non-clustered indexes

Cons

- Conflicts when merging different databases
- Integer can run out of numbers (~2 billion)

Clustered Index On A GUID Column

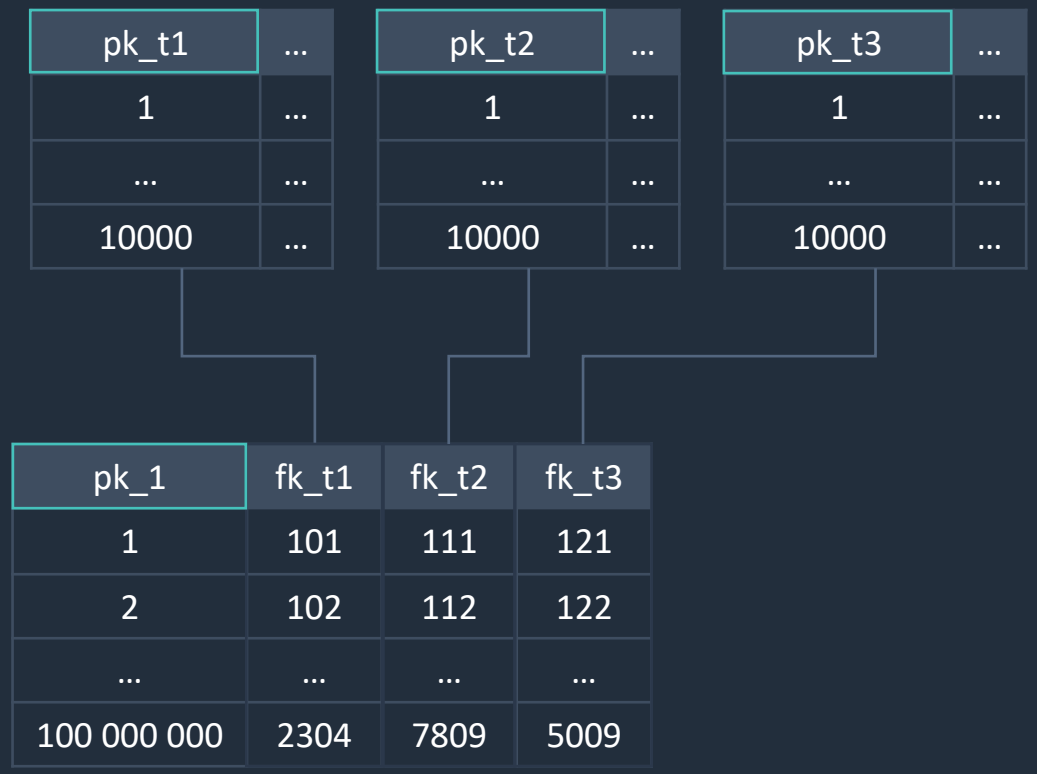
Pros

- Unique across databases and systems
- Easier database merging
- Can be generated offline

Cons

- 16 bytes long
- Bigger fragmentation (not relevant for NEWSEQUENTIALID)
- Slower DML operations

Clustered Index And Its Effects On Table Size



KEY	INT	BIGINT	GUID
pk_t1, pk_t2, pk_t3	39 KB	78 KB	156 KB
pk_1	381 MB	762 MB	1.5 GB
pk_1 + fk_t1 + fk_t2 + fk_t3	1.5 GB	3.1 GB	6.1 GB

Nonclustered Primary Keys

- Primary key uniquely identifies each record within a table
- Clustered index provides efficient retrieval of data

Large results set queries
(typically on low-selectivity columns)

Clustered Index (state, customer_id)

state	customer_id (PK)	name	...
IL	1	Oliver Sanderson	...
NJ	2	Rebecca Newman	...

Range queries

Clustered Index (order_date, order_id)

order_date	order_id (PK)	customer_id	...
2024-01-01	1	276	...
2024-01-01	2	125	...

Master-details pattern when the detail's
ID is unique

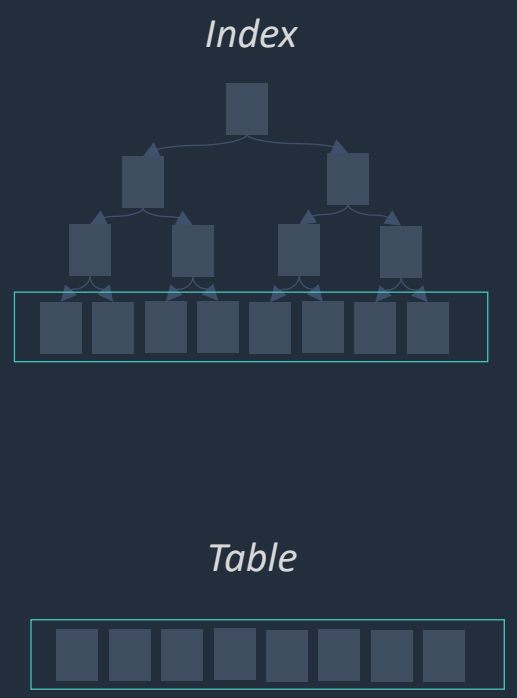
Clustered Index (order_id, order_item_id)

order_id (FK)	order_item_id (PK)	product_id	...
1	1	PRD0047	...
2	2	PRD0047	...

Data Access Operators

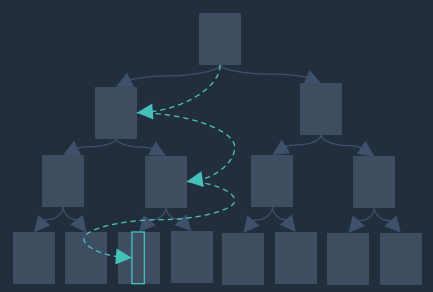
Data Access Operators – Scan Operator

- Iterates over all the rows of its source
- Implemented by
  - Clustered Index Scan
  - Index Scan
  - Table Scan
- Always expensive

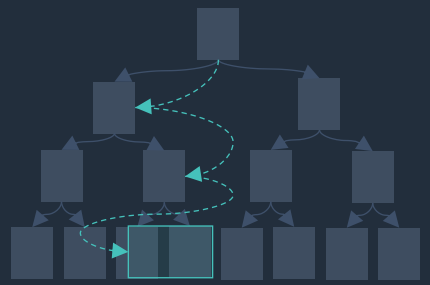


Data Access Operators – Seek Operator

- Seek operator can locate a single row or a range of rows
- Implemented by
  - Clustered Index Seek
  - Index Seek



OR

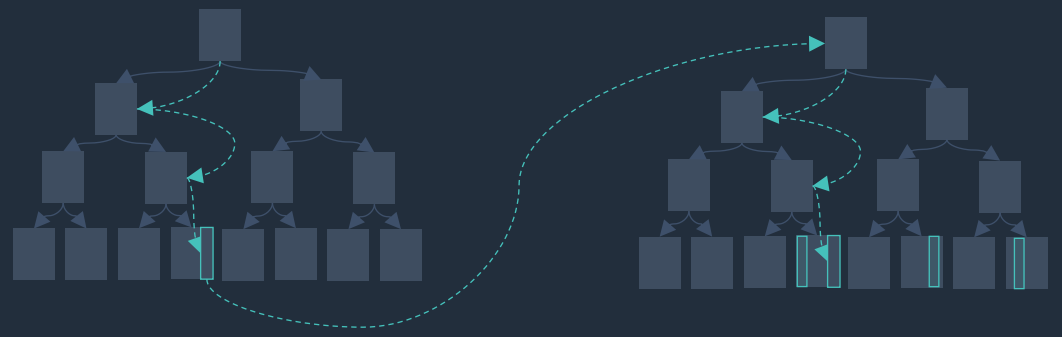


- Works only with B-Tree-based data sources
- Can handle composite indexes

- The most efficient among the operators

Data Access Operators – Bookmark Lookup Operator

- Uses bookmarks to locate rows
- Is part of 2 step data retrieval process
- Implemented by
  - Key Lookup
  - RID Lookup
- Expensive for not selective queries



Query Patterns

Query Patterns – Clustered Index

Fetching all the data without filtering

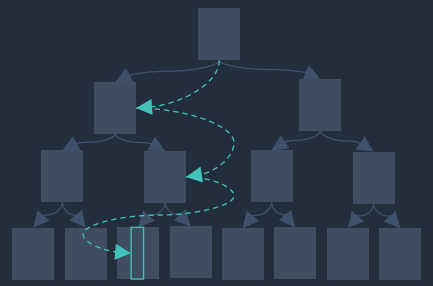
```
SELECT *
FROM customer
```

Index Scan

Filtering by the clustered index column

```
SELECT *
FROM customer
WHERE customer_id = 1
```

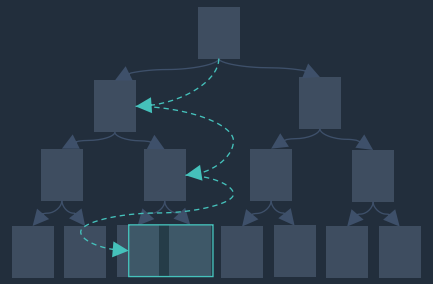
Index Seek



Filtering by the clustered index column (range condition)

```
SELECT *
FROM customer
WHERE customer_id >= 1
  AND customer <= 10
```

Index Seek (Range)

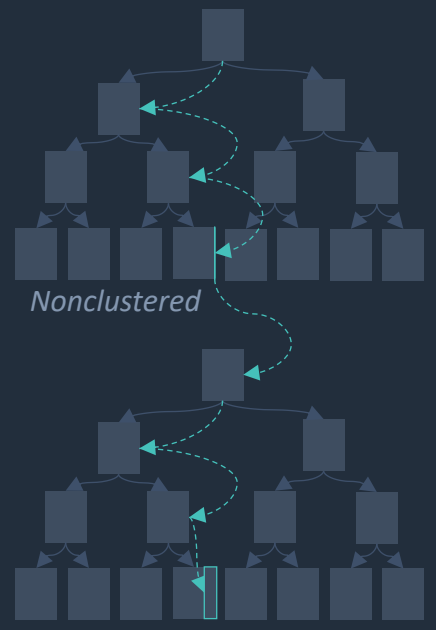


Query Patterns – Nonclustered Unique Index (Singleton)

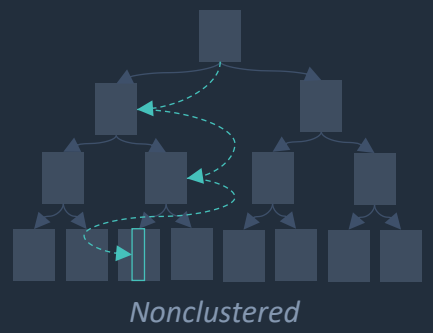
Filtering by the indexed column (unique nonclustered index)

```
SELECT
   first_name,
   last_name
FROM customer
WHERE email = 'd.bond@mail.com'
```

Without covering index -> Index Seek & Key Lookup



With covering index -> Index seek



Query Patterns – Nonclustered Index (Data Range)

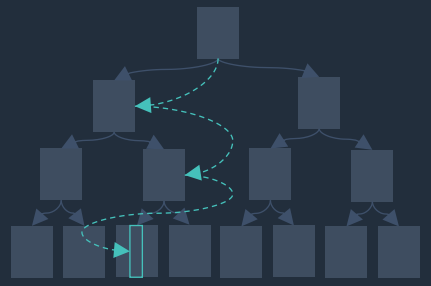
Filtering by the indexed column (nonclustered UNIQUE index)

```
SELECT
   first_name,
   last_name
FROM customer
WHERE email LIKE 'a.%'
```

Filtering by the indexed column (nonclustered NONUNIQUE index)

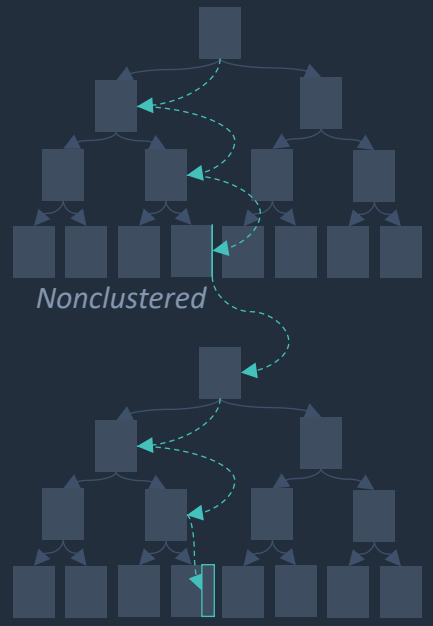
```
SELECT
  customer_id
FROM customer
WHERE state = 'NY'
```

With covering index -> Index seek



Nonclustered

Selective Condition -> Index Seek & Key Lookup



Nonclustered

Clustered

Non-selective condition -> Index scan

Clustered

Query Patterns - Composite Index

Filtering by a column that is in first place in a composite clustered index

```
SELECT product_id
FROM order_item
WHERE order_id = 2
  AND item_id = 1
```

order_id	item_id
1	1
1	2
1	3
2	1
2	2
3	1
3	2
…	

Filtering by composite indexed columns (equality and range conditions)

```
SELECT order_id
FROM order
WHERE status = OPEN
  AND order_date >= 2024-01-01
  AND order_date <= 2024-02-01
```

Composite index on (order_date, status)

…	
2023-12-01	OPEN
2024-01-01	CLOSED
2024-01-02	OPEN
2024-01-03	SHIPPED
2024-01-04	OPEN
2024-02-01	SHIPPED
2024-02-02	OPEN
…	

Composite index on (status, order_date)

…	
OPEN	2023-12-01
OPEN	2023-12-01
OPEN	2024-01-02
OPEN	2024-01-04
OPEN	2024-02-02
SHIPPED	2024-01-01
…	

Query Anti-Patterns – Non-sargable queries

Filtering by a non-indexed column

```
SELECT *
FROM customer
WHERE first_name = 'Bob'
```

Applying a function to a filtering field

```
SELECT *
FROM customer
WHERE LOWER(email) = 'h.bower@mail.com'
```

Comparison to strings with a leading wild character

```
SELECT *
FROM customer
WHERE email LIKE '%a'
```

Applying a mathematical operation to a filtering field

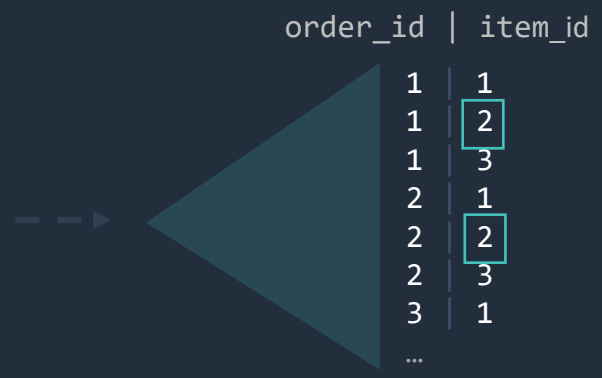
```
SELECT *
FROM product
WHERE product_price * 2 = 1000
```

Filtering on an indexed column with low cardinality

```
SELECT *
FROM customer
WHERE state = 'NJ'
```

Filtering on a column that is not in the first place in a composite index (Composite index on order_id, item_id example)

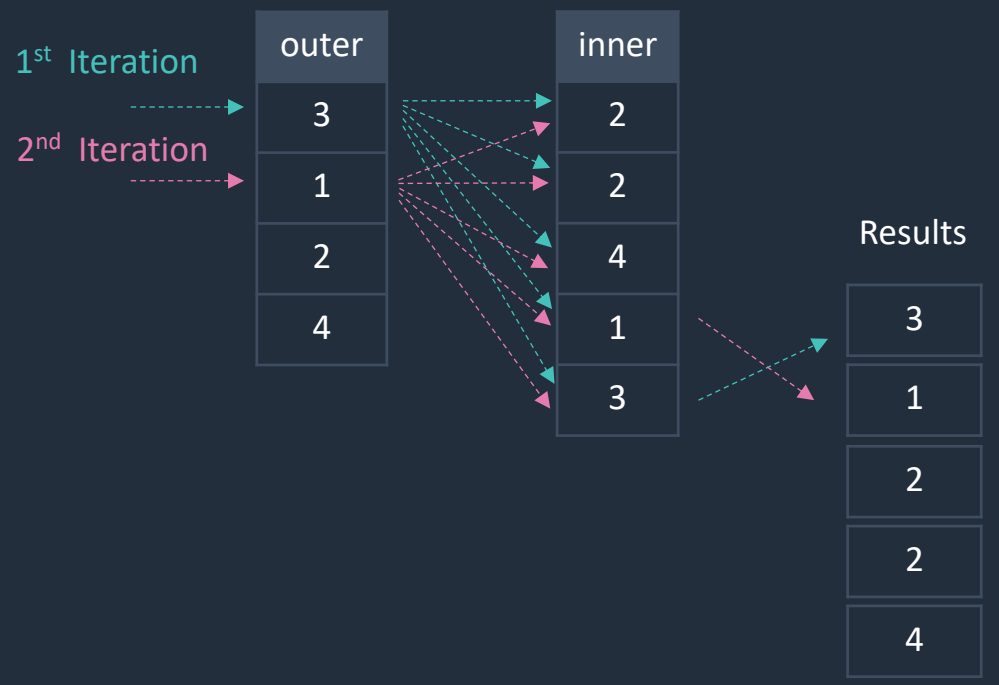
```
SELECT *
FROM order
WHERE item_id = 2
```



Join Algorithms

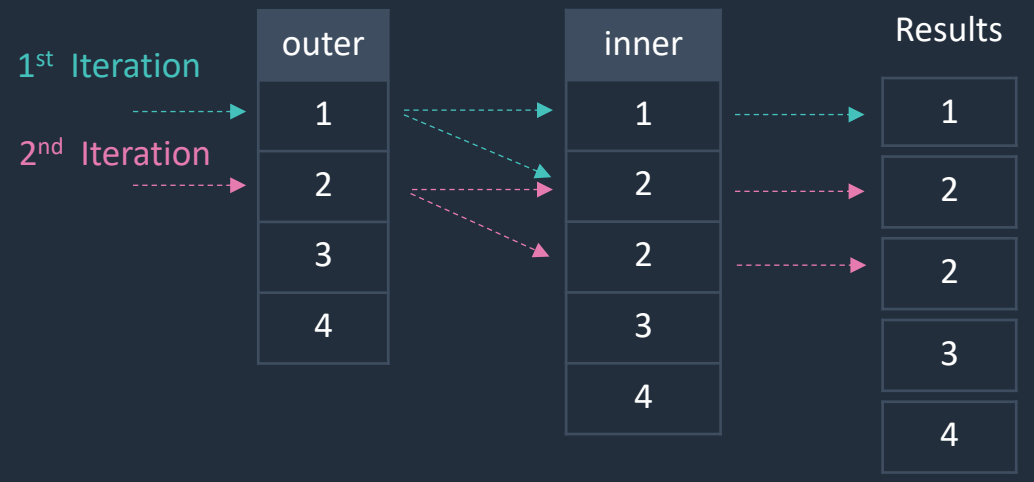
Nested Loops

- Most common type of a JOIN
- The inner table is processed for each row from the outer table
- Suitable for small tables
- Performance can degrade significantly for large tables
- Small memory footprint
- Used with Key Lookups
- Preserves the order of the outer table



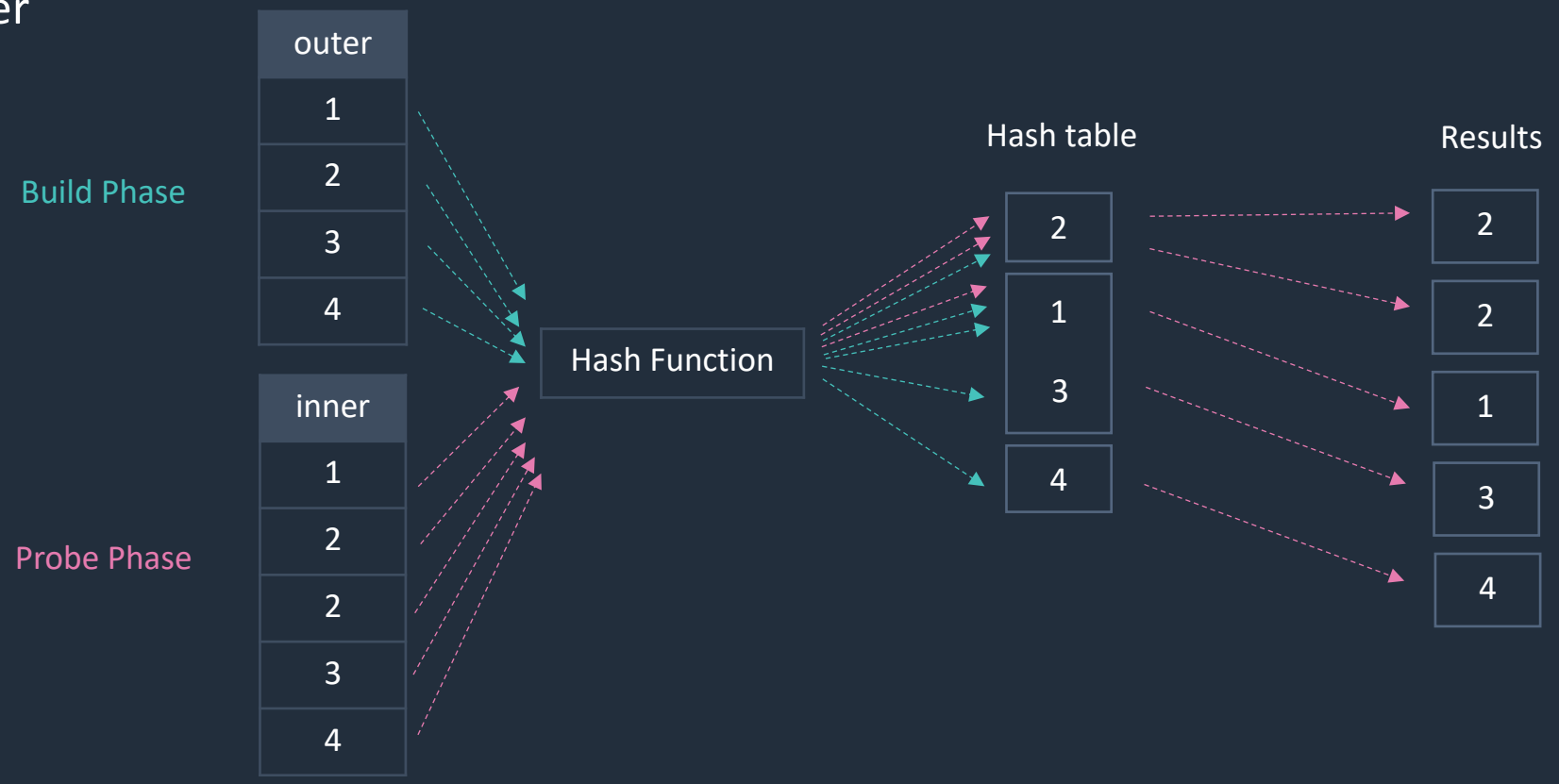
Merge Join

- Requires both inputs to be sorted
- Efficient for large datasets
- Requires additional memory to buffer results
- Preserves the order of the input rows



Hash Join

- Requires memory to build a hash table
- Efficient for large datasets
- Does not preserve the input order



Foreign Key

Foreign Key – Section Overview

01 Introduction

02 Types Of Referenced Tables

03 On Delete, On Update Settings Guidelines

04 Hands-On

Foreign Keys - Introduction

A foreign key is a column or a set of columns in a table that references the primary key or a unique key in another table

- Used to enforce referential integrity
- If indexed, improve:
  - JOINs
  - On update/delete actions

Types Of Referenced Tables

A lookup table in a one-to-many relationship

status

name	status_id
OPEN	1
SHIPPED	2

order

status_id (FK)	order_id
1	723
2	86

A supertype table in a supertype-subtype relationship (one-to-one)

product

...	product_id
...	46
...	27

hardware_product

hardware_product_id (FK)	disk_size
46	500
27	100

A master table in a master-detail relationship (one-to-many)

order

...	order_id
...	1
...	1

order_item

order_id (FK)	item_id	...
1	1	...
1	2	...

An associative table

customer

customer_id
1
2

customer_wishlist

customer_id (FK)	product_id (FK)
1	22
2	23

product

product_id	...
22	...
23	...

On Delete, On Update Settings

- Cascade
- Set Null
- Set Default
- No Action (default)

```
ALTER TABLE <tablename>
ADD [CONSTRAINT <constraintName>]
FOREIGN KEY REFERENCES <referenceTable> (<referenceColumns>)
[ON DELETE <NO ACTION | CASCADE | SET NULL | SET DEFAULT> ]
[ON UPDATE <NO ACTION | CASCADE | SET NULL | SET DEFAULT> ]
```

On Delete, On Update Settings Guidelines

	Delete/Update
Lookup table	No Action – when a child table existence makes no sense without the foreign key value Set Null – when the foreign key value is optional
Supertype table	Cascade
One-to-many relationship	Cascade – for master-detail relationship Set Null – for important data we do not have full control of (e.g., customer, orders) No Action – for other important data
Many-to-many association of two tables	Cascade

Normalization

Normalization – Section Overview

01 Normalization And Anomalies In Relational Model

02 1st – 4th Normal Forms

03 How Far To Normalize

04 Hands-On

Normalization And Anomalies In Relational Model

Definition

Normalization is a process that assigns attributes to entities so that data redundancies are reduced or eliminated

Anomaly Types

- Insertion Anomaly
- Deletion Anomaly
- Update Anomaly

Insertion anomaly

order_id	product_id	product_name
1	11	tablet
2	12	laptop

Deletion anomaly

order_id	product_id	product_name
1	11	tablet
2	12	laptop

Update anomaly

order_id	product_id	product_name
1	11	tablet
2	12	laptop

order_id	product_id	product_name
1	11	laptop
2	12	laptop

1st Normal Form

Characteristics

- The table has a primary key
- The values in each column are atomic
- The table has no repeating groups

order_id	product
1	laptop, tablet
2	laptop

order_id	product_1	product_2
1	laptop	tablet
2	laptop	NULL

Before normalization

After normalization

order_id	product
1	laptop
1	tablet
2	laptop

2nd Normal Form

Characteristics

- The table is in 1st normal form
- Each non-key column is functionally dependent on the entire primary key
- Aims to remove partial dependencies

PK

order_id	product_id	product_category
1	3	Hardware
1	5	Software
2	3	Hardware

Before normalization

After normalization

order_id	product_id
1	3
1	5
2	3

product_id	product_category
3	Hardware
5	Software

3rd Normal Form

Characteristics

- The table is in 2nd normal form
- Each non-key column depends only on the primary key
- Aims to eliminate transitive dependencies

PK

order_id	product_id	quantity	discount
1	3	1	0
2	3	2	30
3	5	1	0

Before normalization

After normalization

order_id	product_id	quantity
1	3	1
2	3	2
3	5	1

quantity	discount
1	0
2	30

Boyce-Codd Normal Form (BCNF)

Characteristics

- The table is in 3rd normal form
- Every determinant in a table is a candidate key
- Aims to eliminate dependencies between key and non-key columns
- No dependency-preserving BCNF decomposition of every schema
- Violated if there are two candidate keys with overlapping fields

CK1 CK2

product_id	promotion_type	promotion
1	discount_based	bundle
1	engagement_based	loyalty
2	discount_based	bundle

Before normalization

After normalization

product_id	promotion
1	bundle
1	loyalty
2	bundle

promotion	promotion_type
bundle	discount_based
seasonal	discount_based
loyalty	engagement_based
contents	awareness-based

BCNF vs 2NF

PK

product_id	promotion	promotion_type
1	bundle	discount_based
1	loyalty	engagement_based
2	bundle	discount_based

Violation of 2NF

PK

product_id	promotion_type	promotion
1	discount_based	bundle
1	engagement_based	loyalty
2	discount_based	bundle

Violation of BCNF (2NF satisfied)

4th Normal Form

Characteristics

- The table is in BCNF
- Has no multivalued dependencies

PK

product_id	product_category	product_vendor
1	hardware	new technologies
1	digital_cameras	new technologies
1	hardware	dynamic labs
1	digital_cameras	dynamic labs
2	hardware	dynamic labs
2	laptops	dynamic labs

Before normalization

After normalization

product_id	product_category
1	hardware
1	digital_cameras
2	hardware
2	laptops

product_id	product_vendor
1	new technologies
1	dynamic labs
2	dynamic labs

Normalization – Mental Model

Each normal form aims to resolve different types of dependencies

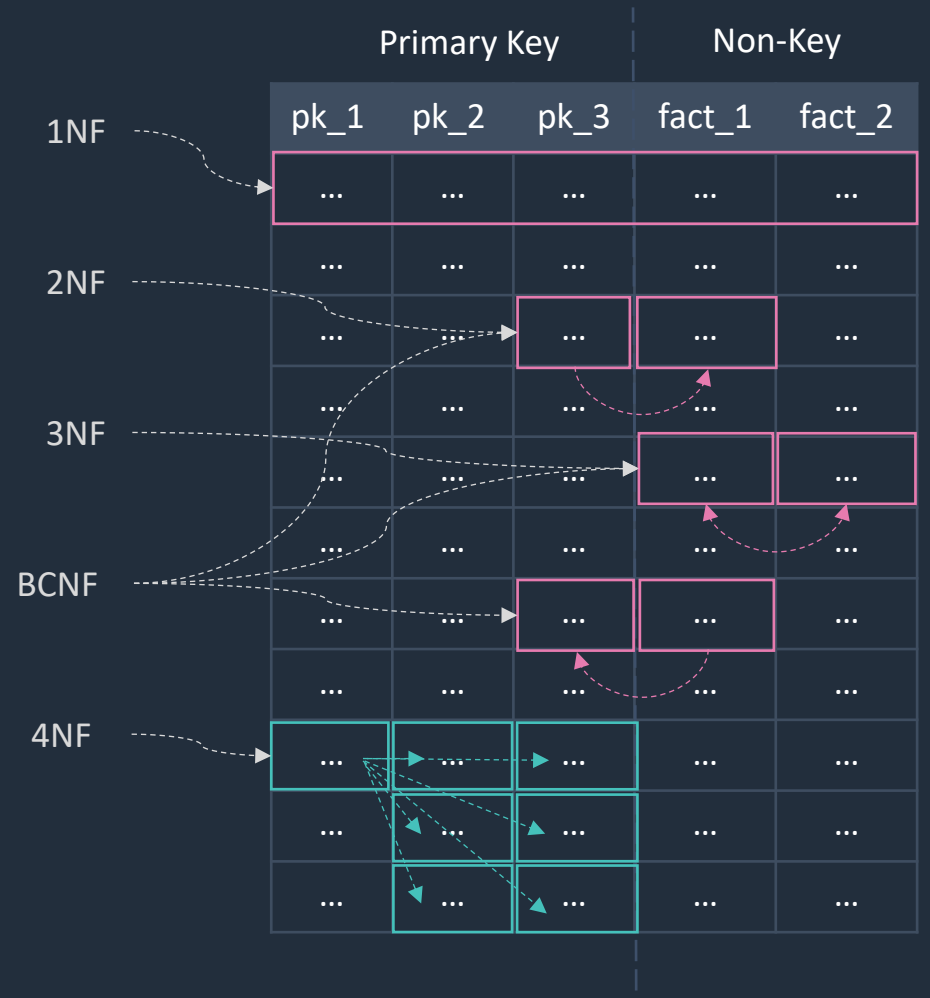
1NF – Repeating groups

2NF – Non-key being dependent on part of the key

3NF – Dependencies between non-key columns

BCNF – Dependencies on non-candidate keys

4NF – A primary key column determines other primary key columns



How Far To Normalize

- The default should BCNF or 3NF
- 2NF or lower if requirements allow

PK

customer_id	name	zip_code	state	city
1	Oliver Sanderson	94583	NC	Fayetteville
2	Rebecca Newman	33134	RI	Providence

PK

customer_id	name	zip_code	state	city
1	Oliver Sanderson	94583	NC	1
2	Rebecca Newman	33134	RI	2

PK

zip_code_id
94583
33134

PK

city_id	state_id	name
1	NC	...
2	RI	...

PK

state_id
NC
RI

How Far To Normalize (continued)

- Lower normal forms for simple requirements

A table not in 1NF

customer_id	options
1	1, 2, 3, 4, 5
…	…

- Storing JSON, or XML as it is if data is accessed together

A table with JSON field

product_id	review_id	content
1	29	{"author_id":662,"author_name":"Samanth…}
…	…	…

Simple Format

Simple Format - Motivation And Implementation

Problem definition

- The customer's phone column contains non-numeric characters

Solution

- Apply a consistent and simple format to the phone column

Advantages

- Simplified querying

```
SELECT
  phone = REPLACE(REPLACE(REPLACE(phone, '-', ''), ')', ''), '(', '')
FROM customer
WHERE customer_id = 1

(299)-087-5966 ➡ 2990875966
```

- Increased data consistency
- Easier debugging and troubleshooting

```
SELECT phone
FROM customer
WHERE customer_id = 1
```

customer_id	phone
1	(090) 123 5678
2	(080) 333 5678
3	090444 1234

customer_id	phone
1	0901235678
2	0803335678
3	0904441234

Lookup Table

Lookup Table – Motivation

Problems with the vedor_name field

- No validation
- Not easy to add new vendors
- Not easy to get a list of vendors

product

product_id	name	vendor_name
45	QuantaSync	Quantum Solutions
55	ZenBlitz	Celestial Group
67	PulseJet	Celestial Group

Check constraint

```
CREATE TABLE product (
  …
  vendor_name VARCHAR (255) NOT NULL
  CHECK (vendor_name IN ('Quantum Dynamics Solutions'…))
);
```

Getting check constraints' metadata

```
SELECT cc.name, cc.definition
FROM sys.check_constraints cc
INNER JOIN sys.tables t ON cc.parent_object_id = t.object_id
WHERE t.name = 'customer'
    AND cc.definition LIKE '(vendor_name IN%'
```

Lookup Table – Pattern Description

Definition

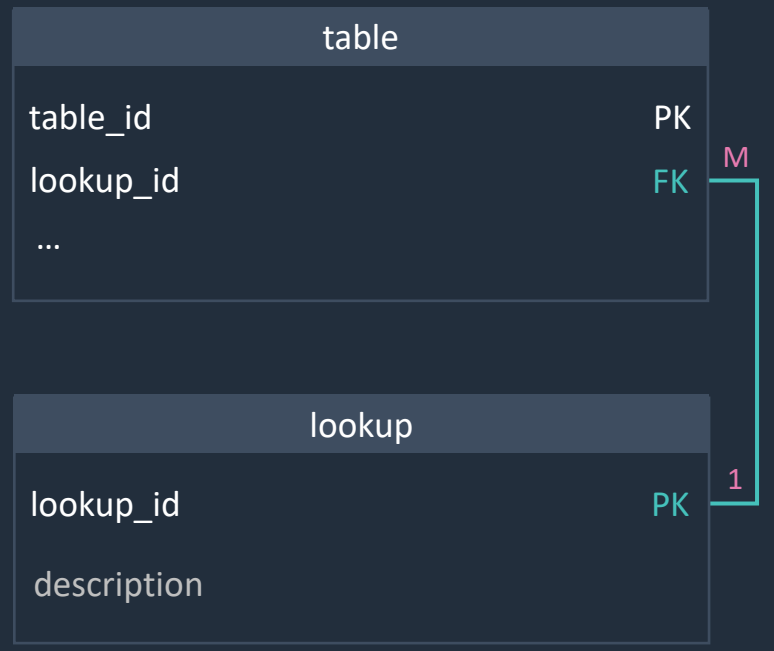
A lookup table is a table used to store predefined values

Benefits

- Data validation
- Provides a single source of truth
- Flexible schema

Drawbacks

- Small negative impact on performance



Lookup Table – Implementation

In the sample database introduction of a vendor lookup table requires the following changes

sample data

product

product_id	name	vendor_name
45	QuantaSync	Quantum Solutions
55	ZenBlitz	Celestial Group
67	PulseJet	Celestial Group
…	…	…

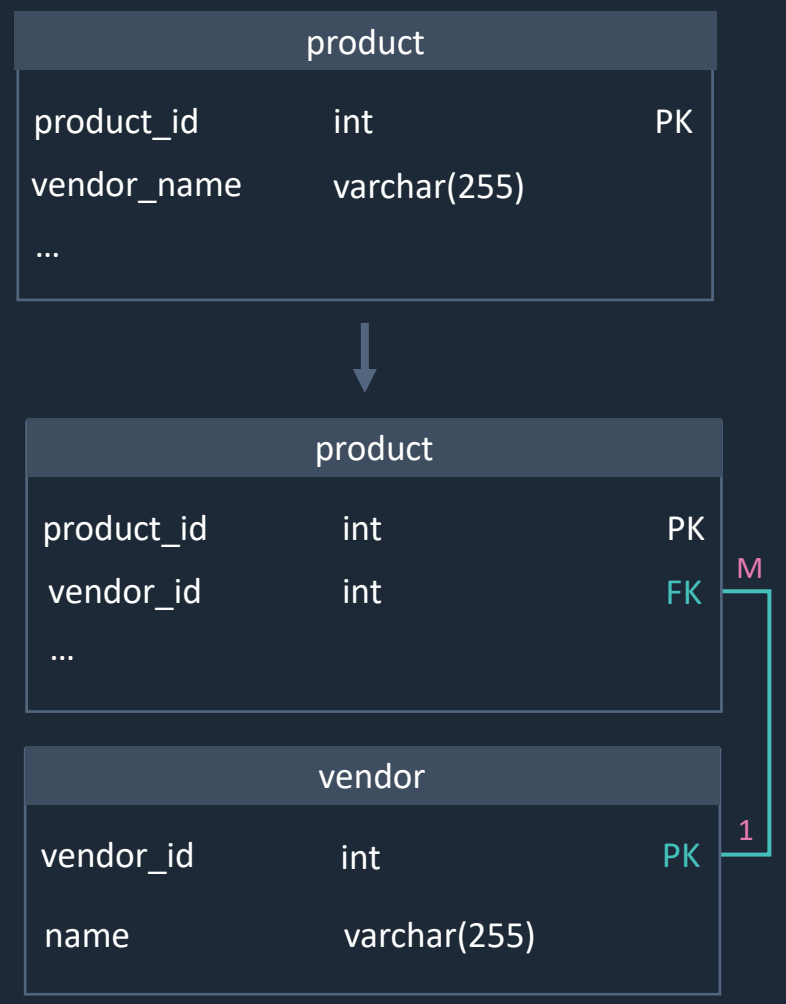
product

product_id	name	vendor_id
45	QuantaSync	1
55	ZenBlitz	2
67	PulseJet	2
…	…	…

vendor

vendor_id	name
1	Quantum Solutions
2	Celestial Group
…	…

data model



Associative Table

Associative Table – Motivation

Current design problems and limitations

- The *wishlist* column in the customer table stores multiple product names

customer

customer_id	wishlist
1	Quantumize, FluxPulse
2	FluxPulse

- No data validation

wishlist
Quantumize, FluxPulse

→

wishlist
Quantumize%%%Pulse

- Unnecessary complex querying

Associative Table – Pattern Description

Definition

An associative table is a table that manages many-to-many relationships between two tables

Benefits

- Improved data integrity
- Minimized data redundancy
- Flexible schema

Drawbacks

- Small negative impact on performance

sample data

customer

customer_id
1
2

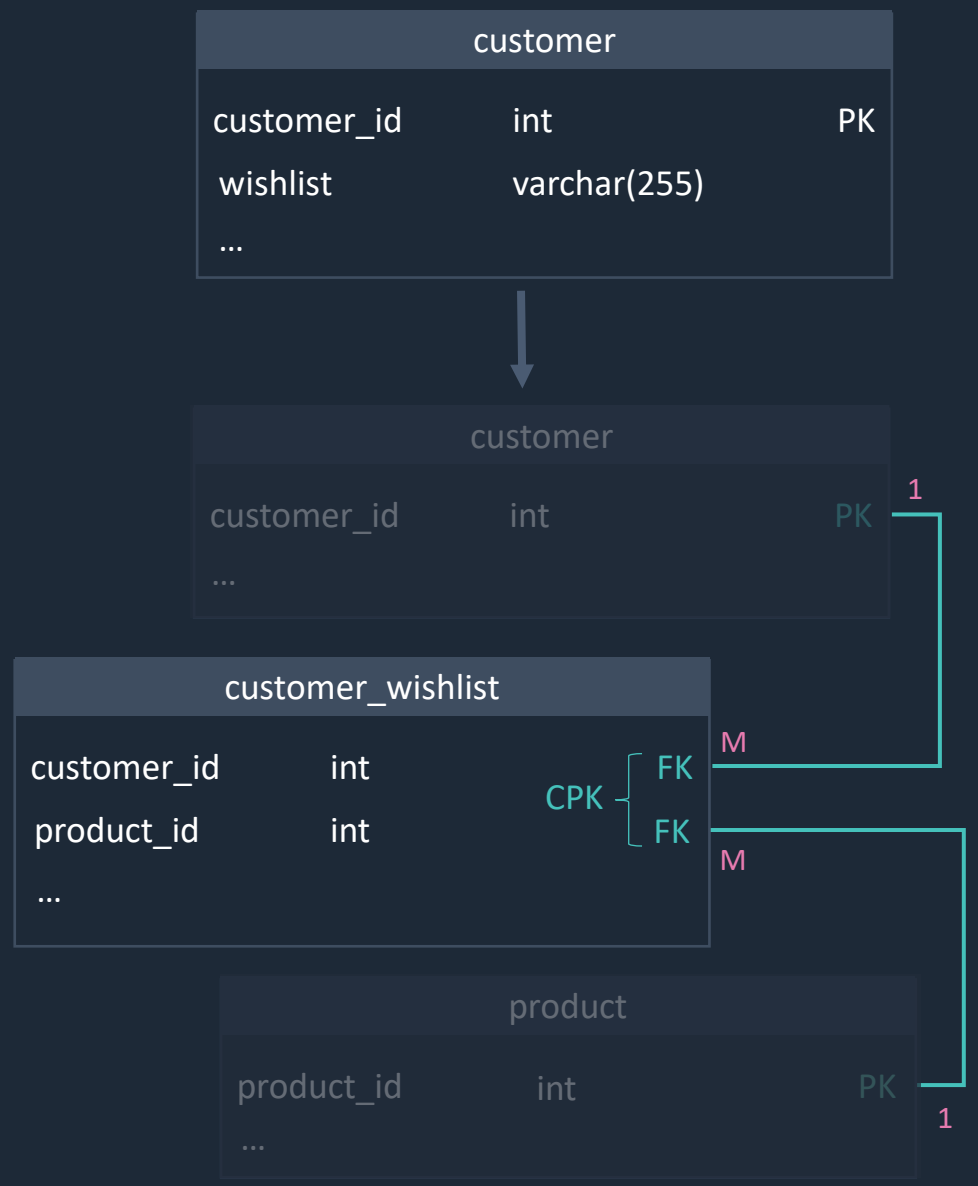
customer_wishlist

customer_id	product_id
1	22
1	23
2	24

product

product_id	name
22	Quantumize
23	FluxPulse

data model



Master-detail

Master-detail – Motivation

Current design limitations

- A record is created in the *order_shipment* table only after shipping all the order items

order_shipment

order_shipment_id	order_id	price	created_at
1	10	100	2024-01-17
2	11	50	2024-02-01
…	…	…	…

Master-detail – Pattern Description

Definition

Master-detail tables are tables in a relationship where one table is an inseparable part of the other

Benefits

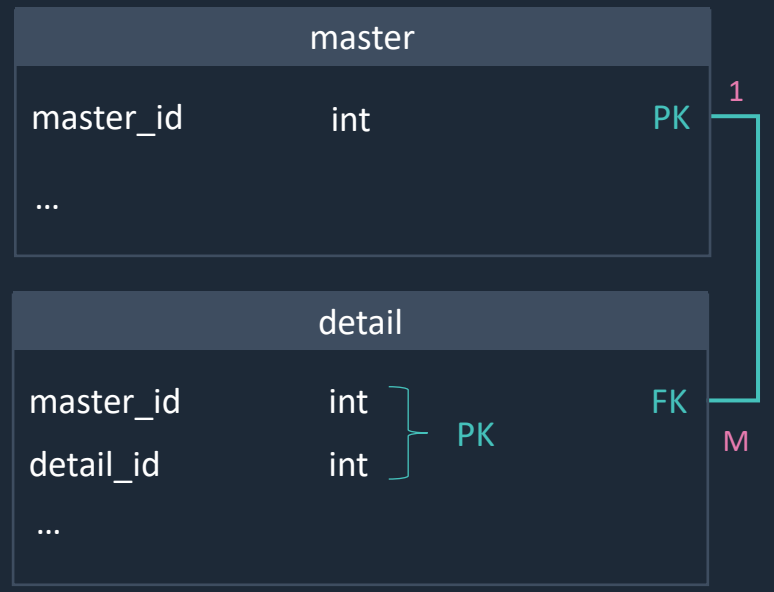
- Improved data integrity
- Simple querying
- Clear data representation

sample data

...	master_id
...	1
...	2

master_id	detail_id	...
1	1	...
1	2	...
1	3	...
2	1	...
2	2	...
2	3	...

data model



History Table

History Table – Section Overview

01 Theory

02 Hands-On (effective date)

03 Hands-On (effective start/end dates)

History Table - Theory

Definition

A history table is a table used to store information about data changes over time

Two implementations

Single-column

promotion

product_id	discount	effective_date
1	20	2024-01-01
1	25	2024-02-01
1	30	2024-04-01

- Simpler data inserts

Two-column

promotion

product_id	discount	effective_start_date	effective_end_date
1	20	2024-01-01	2024-02-01
1	25	2024-02-01	2024-03-01
1	30	2024-04-01	NULL

- Easier data querying and maintenance
- Support for overlaps and gaps

Horizontal Partitioning

Horizontal Partitioning – Section Overview

01 Definition

02 Horizontal Partitioning Types

03 Horizontal Partitioning – Pros And Cons

04 Partition Function

05 Partition Scheme

06 Partitioned Table

07 Aligned And Nonaligned Indexes

08 Merge, Split And Switch Operations

Definition

Horizontal partitioning is a design technique where a large table is divided into smaller, subsets called partitions based on some criteria

- Transparent to applications
- Implemented at the disk level
- Data is divided by a partitioning function based on a partition key
- Requires management

order_id	order_date	product_id	quantity
1102	2022-01-21	PRD0002	1
1103	2022-01-22	PRD0024	2
1104	2023-02-03	PRD0031	1
1105	2023-02-04	PRD0072	2

2022

order_id	order_date	product_id	quantity
1102	2022-01-21	PRD0002	1
1103	2022-01-22	PRD0024	2

2023

order_id	order_date	product_id	quantity
1104	2023-02-03	PRD0031	1
1105	2023-02-04	PRD0072	2

Types Of Horizontal Partitioning

- Range Partitioning
  - Time-series data
  - Historical data archiving
- List Partitioning
  - Regional data
  - Customer segmentation
- Hash Partitioning
  - Load balancing

Data partitioned by year

2022

date	...
2022-01-01	...
2022-01-02	...
2022-01-03	...

2023

date	...
2023-01-01	...
2023-01-02	...
2023-01-03	...

2024

date	...
2024-01-01	...
2024-01-02	...
2024-01-03	...

Data partitioned by state

AL

customer_id	...
24	...
2	...
32	...

...

WY

customer_id	...
3	...
245	...
77	...

Data partitioned by customer ID hash

03

customer_id_hash	...
03ac674216f3...	...
03bc124116a3...	...
03d223ee1671...	...

...

ff

customer_id_hash	...
ffac674216f3...	...
ffbc124176a3...	...
ffd213ee1671...	...

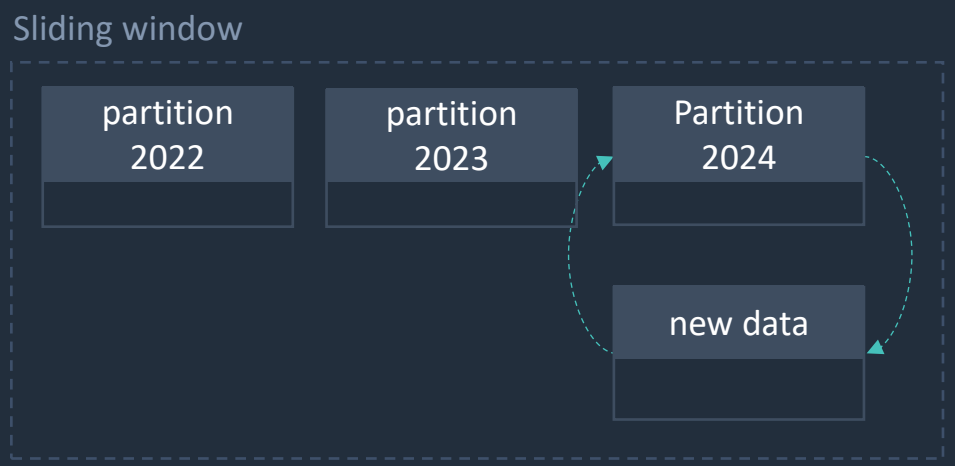
Horizontal Partitioning – Pros And Cons

Pros

- Fast data loading and unloading (e.g., sliding window)
- Faster backup and restore
- Reduced locking
- Easier index management (rebuild, compression settings)
- Slightly improve performance due to partition elimination
- Filegroups can be set to read-only

Cons

- Increased design complexity
- Possible degradation of performance for the following queries:
  - Queries that do not use partition elimination
  - Queries that use operators such as TOP or MIN/MAX



Partition Function

Partition Function is a database object used to define how the rows of a table or index are mapped into partitions based on the values of a specific column

- Partitioning column (only one)
- Range of values
- Range left, and right parameters
- Multiple partition schemes can share a single function

Partitioning Column

order_id	product_id	quantity	order_date
...	...	...	...
1104	PRD0031	1	2022-01-01
...	...	...	...
2105	PRD0072	2	2023-01-01
...	...	...	...
3119	PRD0072	1	2024-01-01
...	...	...	...

```sql
CREATE PARTITION FUNCTION pf_order_date (DATE)
  AS RANGE (LEFT | RIGHT) FOR VALUES
('2022-01-01', '2023-01-01', '2024-01-01');
```

LEFT

Partition 1	Partition 2	Partition 3	Partition 4
...2022-01-01	...2023-01-01	...2024-01-01	...

RIGHT

Partition 1	Partition 2	Partition 3	Partition 4
...	2022-01-01...	2023-01-01...	2024-01-01...

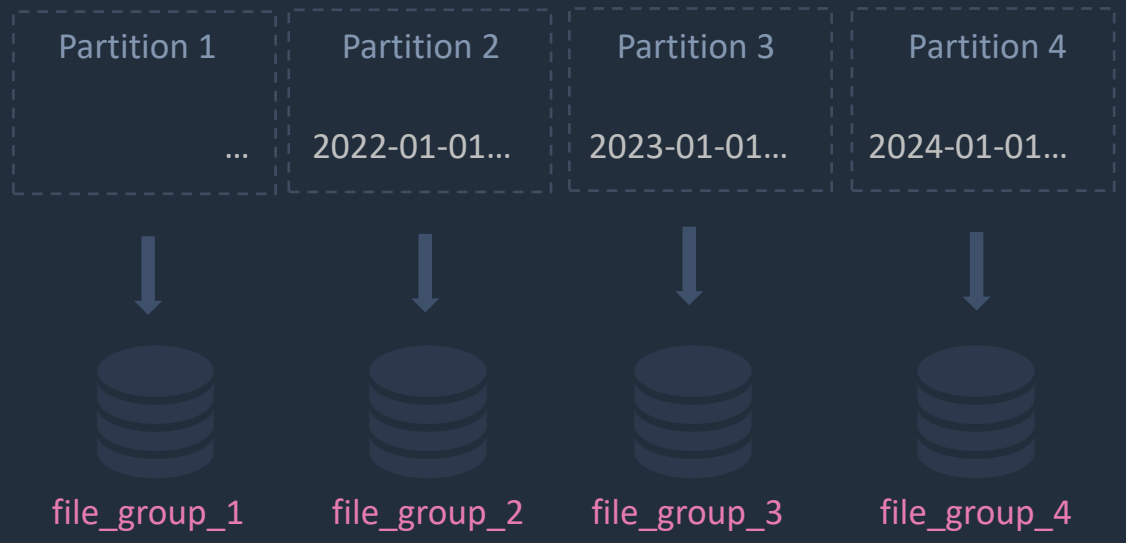
Partition Scheme

A partition scheme is a database object that maps the partitions of a partition function to one or more filegroups

- Different filegroup-partition mappings
- Can use only one partition function
- Can be used by many tables

```sql
CREATE PARTITION SCHEME partition_scheme
   AS PARTITION pf_order_date
TO (file_group_1, file_group_2, file_group_3, file_group_4);
```

```sql
CREATE PARTITION FUNCTION pf_order_date (DATE)
  AS RANGE RIGHT FOR VALUES
('2022-01-01', '2023-01-01', '2024-01-01');
```



Partitioned Table

A partitioned table is a table that has been divided into multiple partitions based on a specified partitioning column and partition scheme

- Has only one partitioning column
- Is created on a partition scheme not on a filegroup
- Each partition has its own clustered index

```
CREATE TABLE order_report (
  order_date DATE NOT NULL,
  ...
  CONSTRAINT pk_order_report PRIMARY KEY (order_date, product_id)
) ON order_time_partitioning_scheme (order_date);
```

Aligned And Nonaligned Indexes

Aligned Index

An aligned index is an index built on the same (or equivalent) partition scheme as its base table

- Efficient queries through partition elimination
- Can be individually maintained
- Allows partition-level operations (SWITCH, TRUNCATE)

Aligned Nonunique NC Index example

	Partition 1	Partition 2	Partition 3
NC Index Key	A B	A C	D E
Clustering key	1 2	3 4	5 6

Nonaligned Index

A nonaligned index is an index that is not partitioned or partitioned differently from their base table

- Allows to enforce uniqueness on non-partition key columns
- More flexibility compared to aligned indexes

Nonaligned Nonunique NC Index example

	PRIMARY
NC Index Key	A A B C D E
Clustering key	1 3 2 4 5 6

Merge And Split Operations

Merge

- Removes one boundary from the partition function and merges two partitions
- If not done properly may cause data movement

partition 1	partition 2	partition 3	partition 4
…	2022-01-01…	2023-01-01…	2024-01-01…

```sql
ALTER PARTITION FUNCTION pf_order_date()
MERGE RANGE ('2022-01-01');
```

partition 1	partition 2	partition 3
…	2023-01-01…	2024-01-01…

Split

- Adds a new boundary to the partition function
- The next filegroup must be specified prior to splitting
- If not done properly may cause data movement

```sql
ALTER PARTITION FUNCTION pf_order_date()
SPLIT RANGE ('2025-01-01');
```

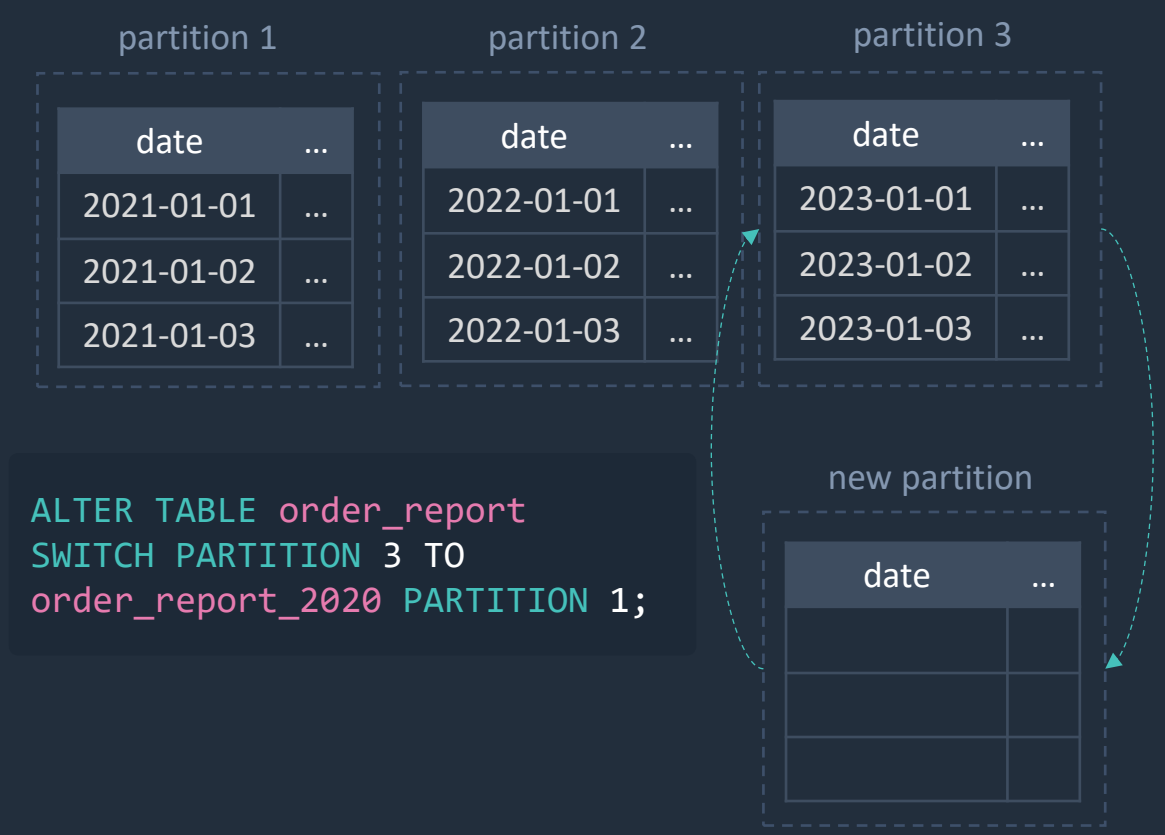
partition 1	partition 2	partition 3	partition 4
…	2023-01-01…	2024-01-01…	2025-01-01…

Switch Operation

- Switches partitions between two tables
- Always a metadata operation – no data movement

Requirements

- All indexes on the table have to be aligned
- Partitions must be on the same column
- Source and target tables must also:
  - Share the same filegroup
  - Have the same column structure and order
  - Have the same clustered and nonclustered indexes
  - Have the same or compatible CHECK Constraint




```sql
ALTER TABLE order_report
SWITCH PARTITION 3 TO
order_report_2020 PARTITION 1;
```

Vertical Partitioning

Vertical Partitioning – Section Overview

01 Data Page

02 Internal Fragmentation

03 External Fragmentation

04 Page Splits

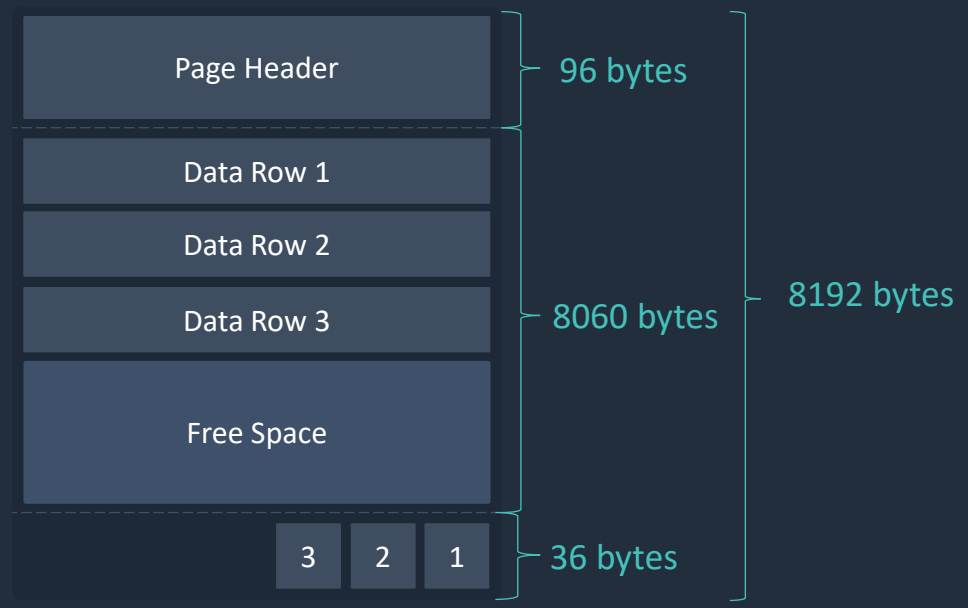
05 Fragmentation Removal And Prevention

Data Page

A data page is a basic unit of storage within a database. It's a fixed-size block of memory allocated to store records or rows of data

Structure

- Page header
- Data rows
- Row offset array
- Free space

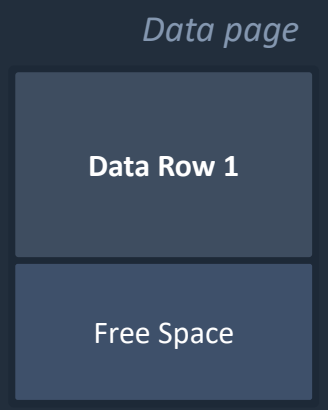


Characteristics

- Compression
- Fill factor
- Ghost records

Internal Fragmentation

Internal fragmentation refers to empty space on data pages



Causes

- Delete operations
- Large data rows
- Update, insert operations causing page splits
- FILLFACTOR, PAD_INDEX parameters

Effects

- More disk space is required to store the same amount of data
- More data has to be read for processing
  (10 000 full pages = 78MB (10 000 x 8KB), 20 000 half-full pages = 156 MB (20 000 x 8KB))

External Fragmentation

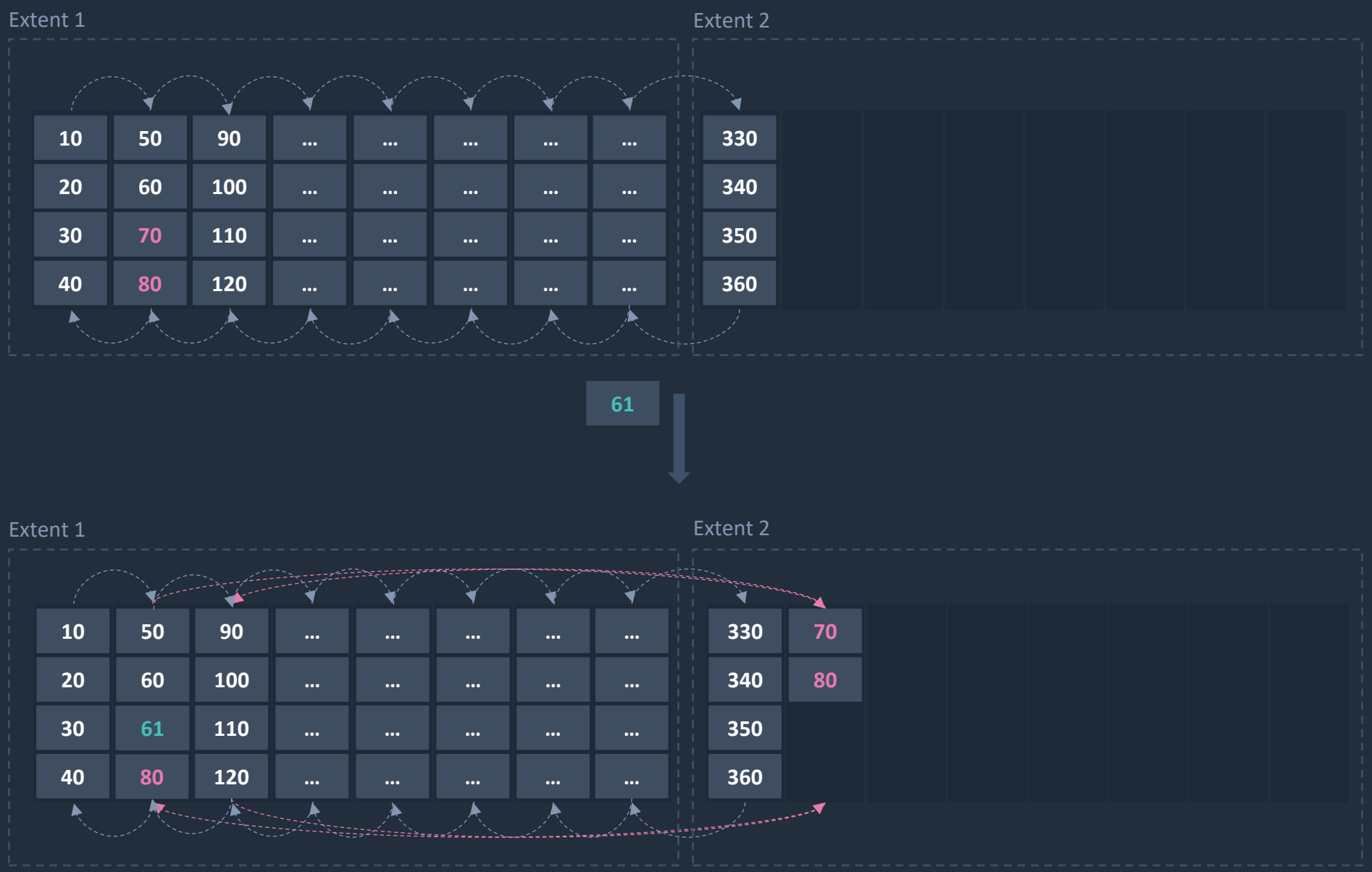
External fragmentation occurs at the file system level, where the physical order of the index pages on disk is not contiguous

Causes

- Not enough contiguous space for allocation
- Page splits

External fragmentation example

Page Splits


Extent 1
Extent 2
10 50 90 ... ...
20 60 100 ... ...
30 70 110 ... ...
40 80 120 ... ...
330
340
350
360
61
Extent 1
Extent 2
10 50 90 ... ...
20 60 100 ... ...
30 61 110 ... ...
40 80 120 ... ...
330 70
340 80
350
360

Fragmentation Removal And Prevention

Removal

- Regular index maintenance (Reorganize, Rebuild, FILLFACTOR)

Prevention

- Use of ever-increasing index key (IDENTITY, newsequentialid)
- Use of static index key
- Use of as small as possible fixed-size data types
- Vertical partitioning
  - Semantics
  - Different access patterns

Modeling Hierarchical Data

Modeling Hierarchical Data – Section Overview

01 Hierarchical Data - Definition

02 Adjacency List Pattern

03 Closure Table Pattern

04 Nested Sets Pattern

05 Path Enumeration Pattern

06 HierarchyID Type

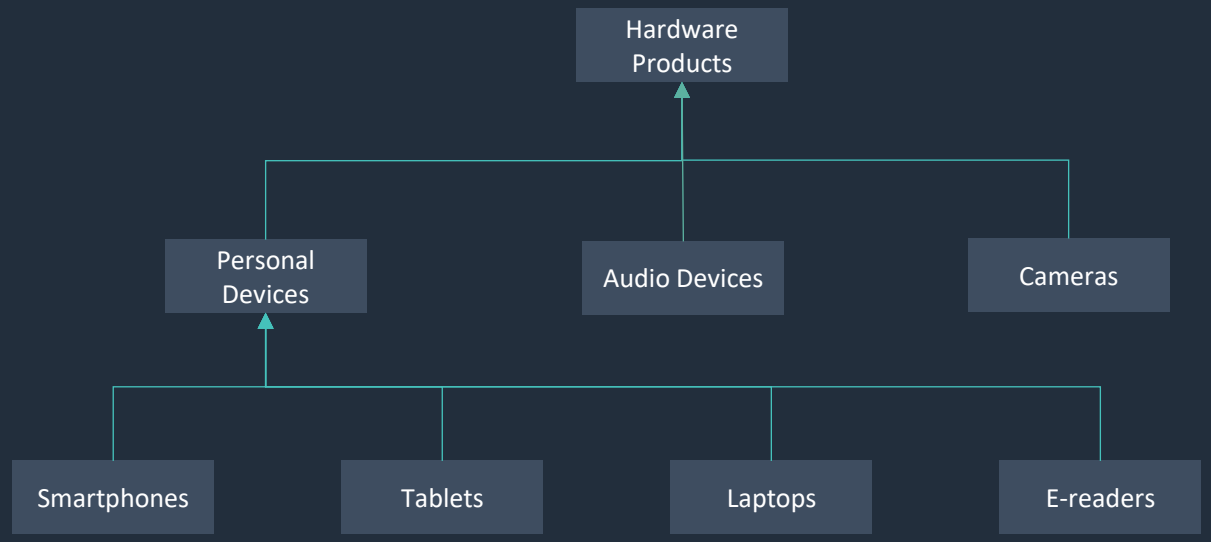
07 HierarchyID - Indexing

Hierarchical Data - Definition

Hierarchical data refers to data that is organized in a hierarchical or tree-like structure

Common patterns for storing hierarchical data

- Adjacency list
- Closure Table
- Nested Sets
- Path Enumeration
- HierarchyID Type (SQL Server only)



Adjacency List

In the Adjacency List pattern, each node in the hierarchy contains a reference to its parent node

Advantages

- Easy to implement
- Easy to maintain
- Referential Integrity

Drawbacks

- Querying requires the use of nested joins or CTE
- Not suitable for deep hierarchies

product_category

product_category_id	name	parent_category_id
1	Hardware Products	NULL
4	Audio Devices	1
14	Headphones	4
43	Gaming Headphones	14

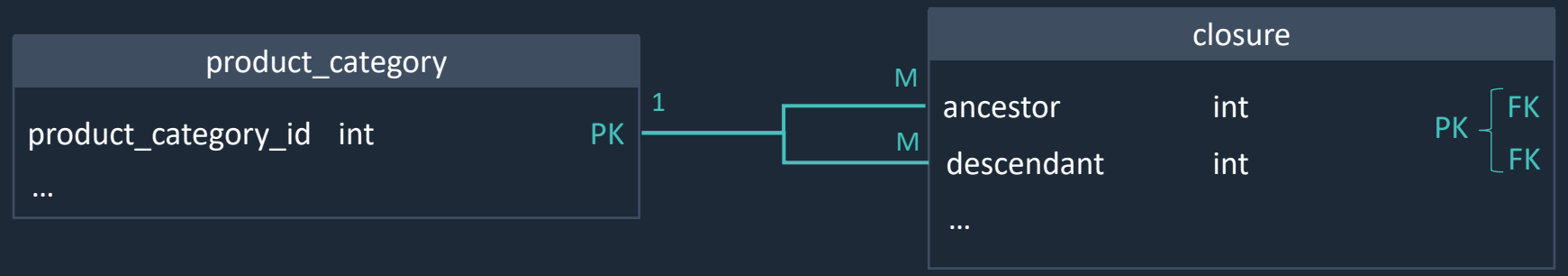
Retrieving parent categories of gaming headphones

```
SELECT
   pc4.parent_category_id,
   pc3.parent_category_id,
   pc2.parent_category_id,
   pc1.parent_category_id
FROM product_category pc1                                  -- Gaming Headphones
LEFT OUTER JOIN product_category pc2
ON pc2.product_category_id = pc1.parent_category_id        -- Headphones
LEFT OUTER JOIN product_category pc3
ON pc3.product_category_id = pc2.parent_category_id        -- Audio Devices
LEFT OUTER JOIN product_category pc4
ON pc4.product_category_id = pc3.parent_category_id        -- Hardware Products
WHERE pc1.product_category_id = 'Gaming Headphones'
```

Closure Table – Structure

A closure table is a table that represents a tree as a list of ancestor and descendant pairs in the hierarchy

data model



sample data

Hardware Products (1)
- -- Personal Devices (3)
- -- Audio Devices (4)
- -- Cameras (5)
- -- Gaming Consoles (6)
- -- Fitness Devices (7)
- -- Other Hardware Products (8)

product_category

product_category_id	name
1	Hardware Products
2	Software Products
3	Personal Devices
4	Audio Devices
5	Cameras
6	Gaming Consoles
7	Fitness Devices
8	Other Hardware Products

closure

ancestor	descendant	depth	valid_from	valid_to
1	1	0	2024-01-01	2024-01-02
1	3	1	2024-01-01	2024-01-02
1	4	1	2024-01-01	2024-01-02
1	5	1	2024-01-01	2024-01-02
1	6	1	2024-01-01	2024-01-02
1	7	1	2024-01-01	2024-01-02
1	8	1	2024-01-01	NULL

Closure Table – Advantages And Drawbacks

Advantages

- Easy to query the data
- Easy to insert/delete nodes
- Referential Integrity

Drawbacks

- Maintenance of additional table is required
- Increased table size

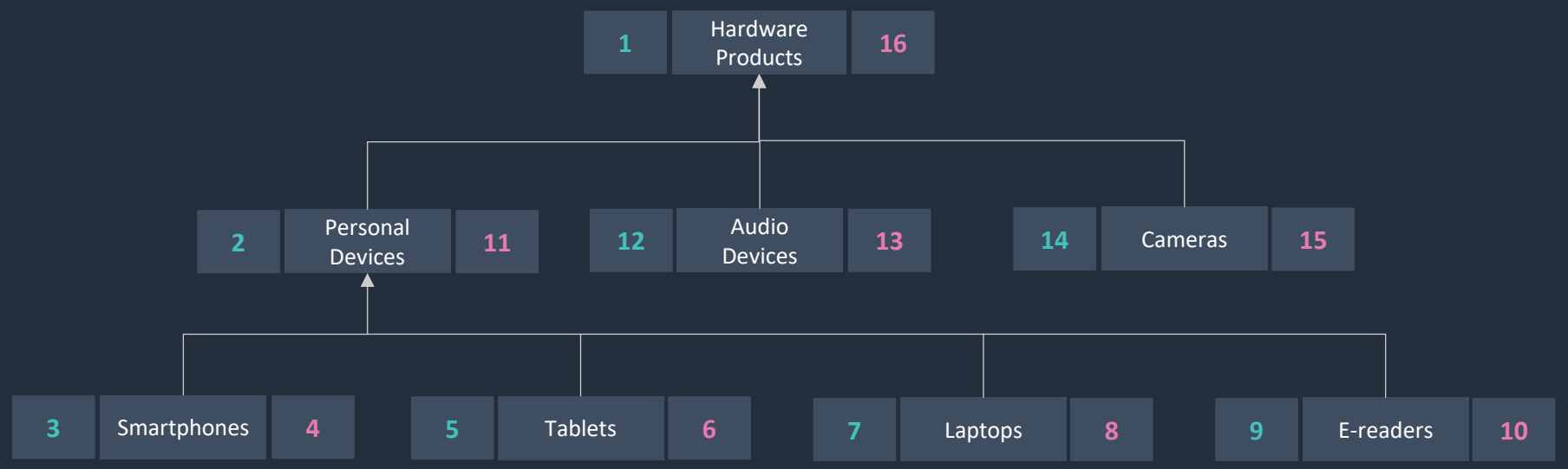
Retrieving subcategories of cameras

```sql
SELECT name
FROM product_category pc
   INNER JOIN product_category_hierarchy pch
   ON pc.product_category_id = pch.descendant
WHERE pch.ancestor = @node_id
```

Nested Sets – Structure

In the Nested Sets pattern, each node in the hierarchy is represented as an interval of numbers

product_category_id	name	left_node_id	right_node_id
1	Hardware Products	1	16
4	Audio Devices	12	13
14	Cameras	14	15
43	Smartphones	3	4



Nested Sets – Advantages And Drawbacks

Advantages

- Easy to query the data

Drawbacks

- Complex updates
- Difficult to understand
- Referential integrity is not enforced by the schema

Retrieving subcategories of cameras

```sql
SELECT name
FROM product_category pc1
   INNER JOIN product_category pc2
   ON pc2.left_id BETWEEN pc1.left_id AND pc1.right_id
WHERE pc1.name = 'Cameras'
```

Adding a node to the hierarchy

```sql
DECLARE @left_parent INT;
SET @left_parent =
    (SELECT left_id FROM product_category WHERE name = 'Hardware Products’);

UPDATE product_category
SET right_id = right_id + 2
WHERE right_id > @left_parent;

UPDATE product_category
SET left_id = left_id + 2
WHERE left_id > @left_parent;

INSERT INTO product_category(product_category_id, name, left_id, right_id)
VALUES (100, 'Photography Accessories', @left_parent + 1, @left_parent + 2);
```

Path Enumeration

In the Path Enumeration pattern, each hierarchy node is assigned a unique path or identifier representing its position

Advantages

- Easy to implement
- Easy to query

Drawbacks

- Data integrity is not enforced by the schema
- If indexed, the path is limited to 1700 bytes

product_category

product_category_id	name	path
1	Hardware Products	1
4	Audio Devices	1|4
14	Headphones	1|4|14
43	Gaming Headphones	1|4|14|43

1 4 14 43

Hardware Products | Audio Devices | Headphones | Gaming Headphones

Retrieving subcategories of audio devices

```
SELECT name
FROM product_category
WHERE product_category.path LIKE '1|4' + '%'
```

Retrieving supercategories of audio devices

```
SELECT name
FROM product_category
WHERE '1|4' LIKE product_category.path + '%'
```

HierarchyID Type (SQL Server only)

HierarchyID is a data type used to represent hierarchical data structures

- Supports sorting
- CLR type
- 892 bytes length limit
- Does not enforce uniqueness automatically
- Provides many built-in methods and functions

id	name	node_binary	node_text	path
1	Hardware Products	0x58	/1/	/1/
3	Personal Devices	0x5AC0	/1/1/	/1/3/
4	Audio Devices	0x5B40	/1/2/	/1/4/

Insert

/1/1.5/

Sort ASC

/1/
/1/1/
/1/1.5/
/1/2/

HierarchyID Methods – IsDescendantOf(node HIERARCHYID)

name	hierarchy
Hardware Products	/1/
Software Products	/2/
Personal Devices	/1/1/
Audio Devices	/1/2/
Headphones	/1/2/1/
Over-Ear Headphones	/1/2/1/1/
On-Ear Headphones	/1/2/1/2/
Gaming Headphones	/1/2/1/3/
Earbuds	/1/2/2/
Cameras	/1/3/
Gaming Consoles	/1/4/
Gaming Station	/1/4/2/
Other Hardware Products	/1/6/

Retrieves ancestors of Headphones

```
@node=('/1/2/1/')

SELECT *
FROM product_category pc
WHERE @node.IsDescendantOf(hierarchy) = 1;
```

name	hierarchy
Hardware Products	/1/
Audio Devices	/1/2/
Headphones	/1/2/1/

Retrieves descendants of Headphones

```
SELECT *
FROM product_category pc
WHERE hierarchy.IsDescendantOf('/1/2/1/') = 1;
```

name	hierarchy
Headphones	/1/2/1/
Over-Ear Headphones	/1/2/1/1/
On-Ear Headphones	/1/2/1/2/
Gaming Headphones	/1/2/1/3/

HierarchyID Methods – GetReparentedValue(old_root, new_root)

name	hierarchy
Hardware Products	/1/
Software Products	/2/
Personal Devices	/1/1/
Audio Devices	/1/2/
Headphones	/1/2/1/
Over-Ear Headphones	/1/2/1/1/
On-Ear Headphones	/1/2/1/2/
Gaming Headphones	/1/2/1/3/
Earbuds	/1/2/2/
Cameras	/1/3/
Gaming Consoles	/1/4/
Gaming Station	/1/4/2/
Other Hardware Products	/1/6/

Moves Headphones descendants to Other Hardware Products

```
UPDATE product_category
  SET hierarchy =
    hierarchy.GetReparentedValue('/1/2/1/', '/1/6/')
WHERE hierarchy.IsDescendantOf('/1/2/1/') = 1
  AND hierarchy.ToString() != '/1/2/1/';
```

name	hierarchy
Other Hardware Products	/1/6/
Over-Ear Headphones	/1/6/1/
On-Ear Headphones	/1/6/2/
Gaming Headphones	/1/6/3/

HierarchyID Methods – GetAncestor(level INT), GetDescendant(child_1, child_2)

name	hierarchy
Hardware Products	/1/
Software Products	/2/
Personal Devices	/1/1/
Audio Devices	/1/2/
Headphones	/1/2/1/
Over-Ear Headphones	/1/2/1/1/
On-Ear Headphones	/1/2/1/2/
Gaming Headphones	/1/2/1/3/
Earbuds	/1/2/2/
Cameras	/1/3/
Gaming Consoles	/1/4/
Gaming Station	/1/4/2/
Other Hardware Products	/1/6/

Returns the parent of Headphones

```
SELECT
 hierarchy.GetAncestor(1)
FROM product_category
WHERE hierarchy = ('/1/2/1/');
```

/1/2/

Generates a new descendant of Headphones

```
SELECT
 hierarchy.GetDescendant('/1/2/1/3/',NULL)
FROM product_category
WHERE hierarchy = ('/1/2/1/');
```

/1/2/1/4/

HierarchyID Other Methods

- GetLevel()
- ToString()
- GetRoot()

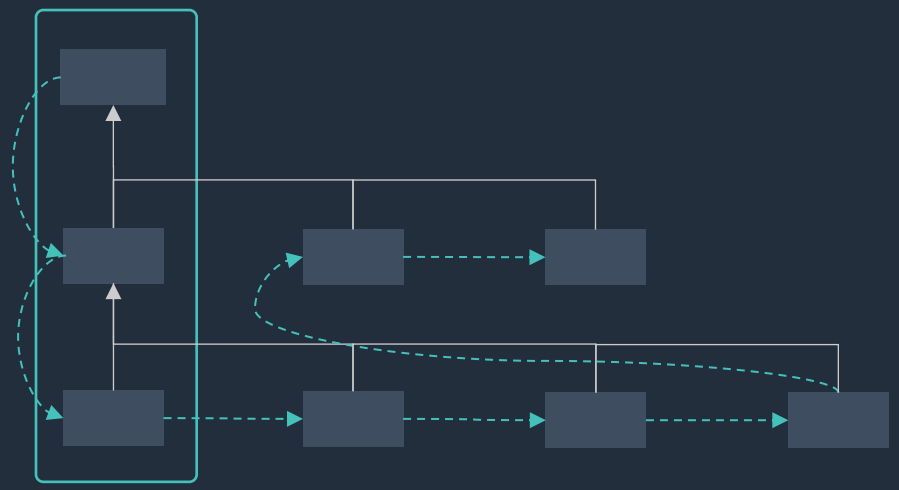
```
('/1/2/4/').GetLevel() -> 3

('0x5B61').ToString() -> '/1/2/4/'

HIERARCHYID::GetRoot() -> /
```

HierarchyID – Indexing

- Depth-first index
- Breadth-first

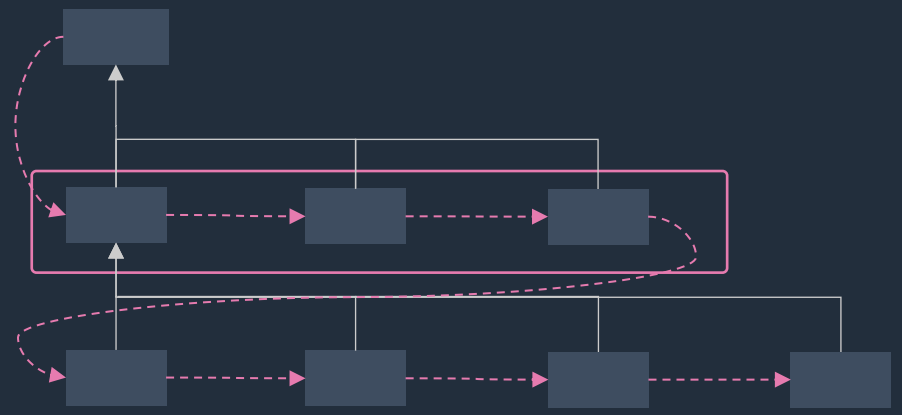
Depth-First

```
CREATE INDEX
   ix_product_category_hierarchy_depth_first ON
production.product_category(hierarchy)
```



Breadth-First

```
CREATE INDEX
   ix_product_category_hierarchy_depth_first ON
production.product_category(level, hierarchy)
```



Modeling Table Inheritance

Inheritance Patterns

Single Table Inheritance

- Single tables contains all data
- Uses a discriminator field

product

id	type	name	weight	required_ram
1	SOFTWARE	Basic English	NULL	500
2	HARDWARE	Laptop	3	NULL

Concrete Table Inheritance

- Each entity has its table
- No referential integrity between the tables

software

id	name	platform
1	Basic English	Windows

hardware

id	name	weight
1	Laptop	3

Class Table Inheritance

- Supertype and subtype entities

product

id	name
1	Basic English
2	Laptop

software_product

id	required_ram
1	500

hardware_product

id	weight
2	3

Handling Multilanguage Data

Handling Multilanguage Data – Section Overview

01 Unicode And Encoding Selection

02 Collations

03 Schema Design

Unicode And Encoding Selection

Unicode

- Multilingual Support
- UTF-8, UTF-16

Pros for UTF-8

- Less memory required for storing ASCII characters
- Fully compatible with ASCII
- Dominates the web (98% web pages use UTF-8)
- Has no byte order

Code Range	UTF-8 (bytes)	UTF-16 (bytes)
0–127	1	2
128 – 159 160 – 1,023 1,024 – 2,047	2	2
2,048 – 16,383 16,384 – 65,535	3	2
65,536 – 262,143 262,144 – 1,114,111	4	4

Pros for UTF-16

- Faster text processing
- Less memory required for storing characters in the range from 2048 to 65535
- Easier integration with programming languages (.NET, Java, JavaScript)

Collation Basics

Defines a set of rules that dictate how data is sorted, compared, and stored

- Can be specified at the instance, database, column, and expression levels
- Affects only textual data
- Collation options
  - Case-sensitivity (_CS, _CI)
  - Accent-sensitivity (_AS, _AI)
  - Kana-sensitive (_KS)
  - Width-sensitive (_WS)
  - Variation-selector-sensitive (_VSS)
- Collation example: Latin1_General_100_CI_AS

Collation Types

Collation	Character based	Binary
Windows	▪ Based on Windows locales ▪ Same sorting and comparison rules for Unicode and non-Unicode ▪ Example: Polish_CI_AS	▪ Different sorting rules for Unicode and non-Unicode ▪ Better performance than character-based ▪ Case-sensitive ▪ Windows Collation example: Polish_BIN, Polish_BIN2 ▪ SQL Collation examples: SQL_Latin1_General_CP437_BIN, SQL_Latin1_General_CP437_BIN2

SQL	▪ Use different sorting rules for Unicode and non-Unicode
	  ▪ non-Unicode: `'-b' < 'a'`
	  ▪ Unicode: `N'-b' > N'a'`
	▪ Maintained for backward compatibility
	▪ Examples: SQL_Latin1_General_CP1250_CI_AS	

Collation Selection

- Supports most users (e.g., case-insensitive)
- Collation that is standardized across the system
- Preferably Unicode-only
- A Windows collation should be the default

Handling Multi-language Data
Schema Design

Column Per Language

Benefits

- Easy to implement
- Requires fewer JOINs comparing to other methods
- A different collation can be set for each column
- Allows full-text search

Drawbacks

- Hard to maintain
- Possible NULL values
- Complex querying

Retrieval of a product

```
SELECT
   product_id,
   CASE @selected_language
     WHEN 'pl-PL' THEN name_pl, description_pl
     WHEN 'ja-JP' THEN name_jp, description_jp
     ELSE name_eng, description_eng
   END
FROM product
WHERE p.product_id = 1
```

product		
product_id	int	PK
name_eng	nvarchar(50)	
name_pl	nvarchar(50)	
name_jp	nvarchar(50)	
description_eng	nvarchar(500)	
description_pl	nvarchar(500)	
description_jp	nvarchar(500)	
...	...	

customer		
customer_id	int	PK
address_eng	nvarchar(100)	
address_pl	nvarchar(100)	
address_jp	nvarchar(100)	
...	...	

Single Translation Table For All Entities

Benefits

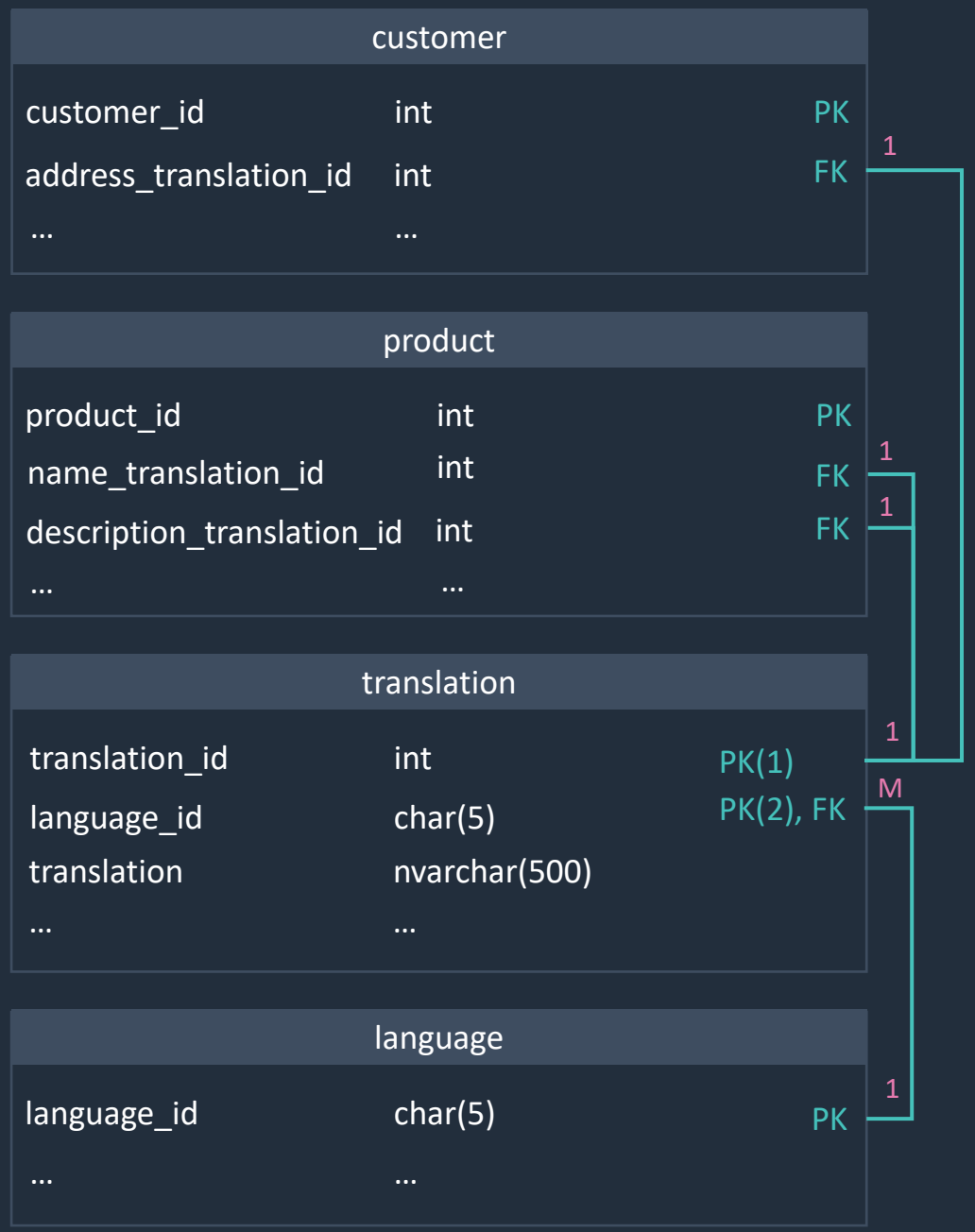
- Simple schema
- Easy to add new translations

Drawbacks

- Single collation for all the languages
- Single table can become a bottleneck
- Difficult to query

Retrieval of a product

```
SELECT *
FROM product p
   JOIN translation t1
       ON p.name_translation_id = t1.translation_id
   JOIN translation t2
       ON p.description_translation_id = t2.translation_id
WHERE p.product_id = 1
  AND t1.language_id = 'ja-JP'
  AND t2.language_id = 'ja-JP'
```



Single Translation Table For Each Entity

Benefits

- Easy to add new translations
- Easy to query (one JOIN)

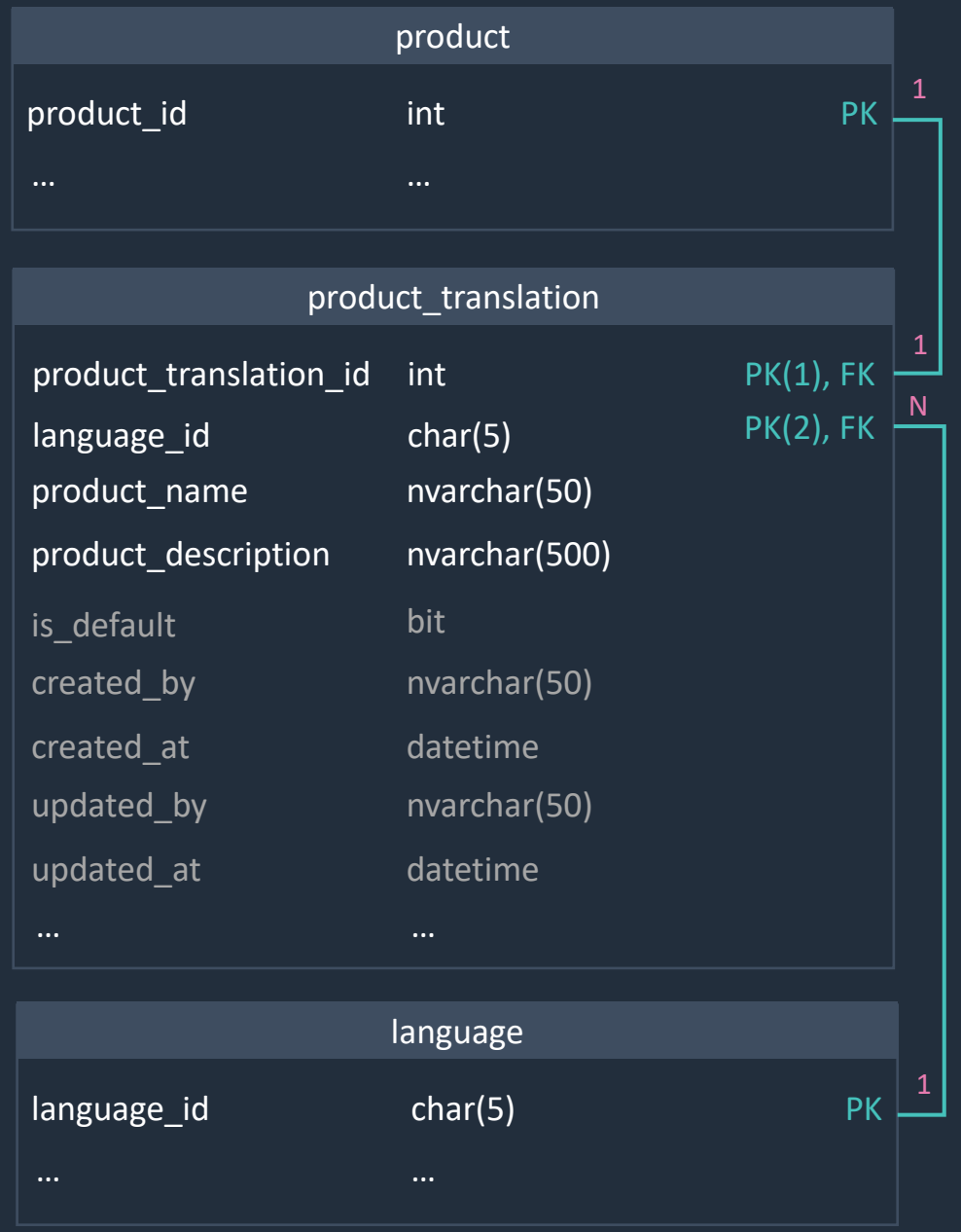
Retrieval of a product

```
SELECT *
FROM product p
   JOIN product_translation pt
       ON p.product_id = pt.product_translation_id
WHERE p.product_id = 1
  AND pt.language_id = 'ja-JP'
```

- Well normalized

Drawbacks

- Increased number of tables
- Single collation for all the languages



Soft Delete

Soft Delete

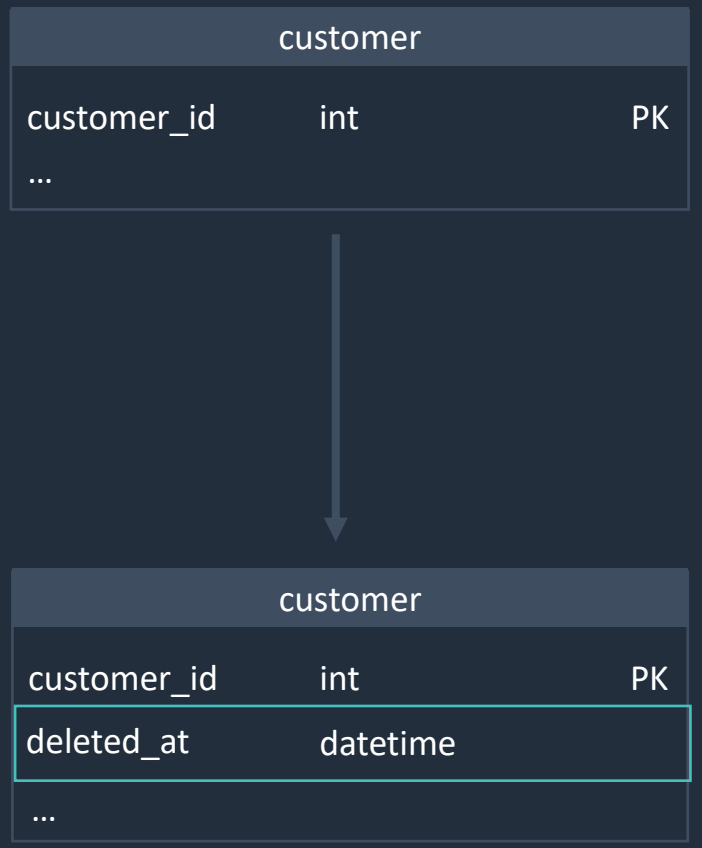
Soft delete refers to flagging a record as “deleted” or inactive instead of permanently deleting it

Benefits

- Allows for easy data recovery
- User error mitigation
- Customer experience
- Auditing and compliance

Drawbacks

- Complexity in queries
- Increased database size



Soft Delete – Additional Notes

Not soft deleted customer
(included in the calculation)

Soft deleted customer
(excluded from the calculation)

customer_id	order_id
1	1
2	2

order_id	item_id	product_id	quantity
1	1	45	1
2	2	27	3
2	3	28	2

product_id	price
45	500
27	300
28	400

Denormalization With JSON

Denormalization With JSON

Denormalization is a technique where relational structures are stored in a denormalized form

Benefits

- Simplified integration and querying
- Fewer joins required

Drawbacks

- Loss of relational integrity
- Potential data redundancy
- Limited querying and indexing

PK

product_id	review_id	content
1	29	{"author_id":662,"author_name":"Samanth...}
1	98	{"author_id":504,"author_name":"Sara ...}
...	...	...

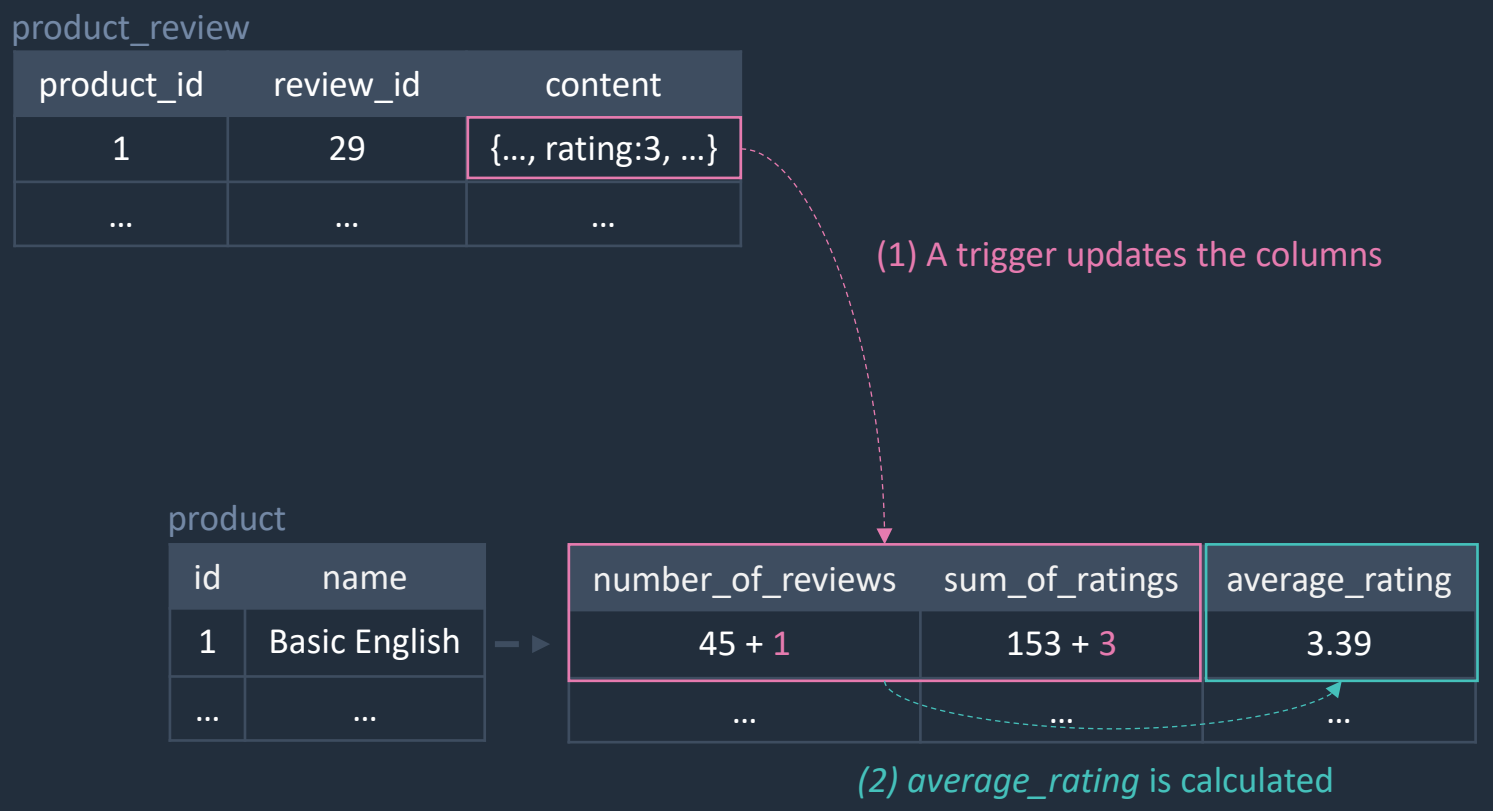
```
{
  author_id: 662,
  author_name: "Samanth  Ellison",
  rating: 4,
  text: "Product received as expected",
  review_date: "2020-01-09T17:35:31",
  images: [ "https://blob.storage/image/3385"  ]
}
```

Computed Column

Computed Column

A computed column is a column whose values are derived or computed based on an expression or formula

- Not physically stored in the database by default
- Can be indexed
  - Determinism
  - Data type



Computed Column – Benefits And Drawbacks

Benefits

- Improved query performance for certain queries
- Dynamic updates

Drawbacks

- Performance Impact
- Maintenance complexity

Sensitive Data Obfuscation

Sensitive Data Obfuscation – Section Overview

01 Use Cases

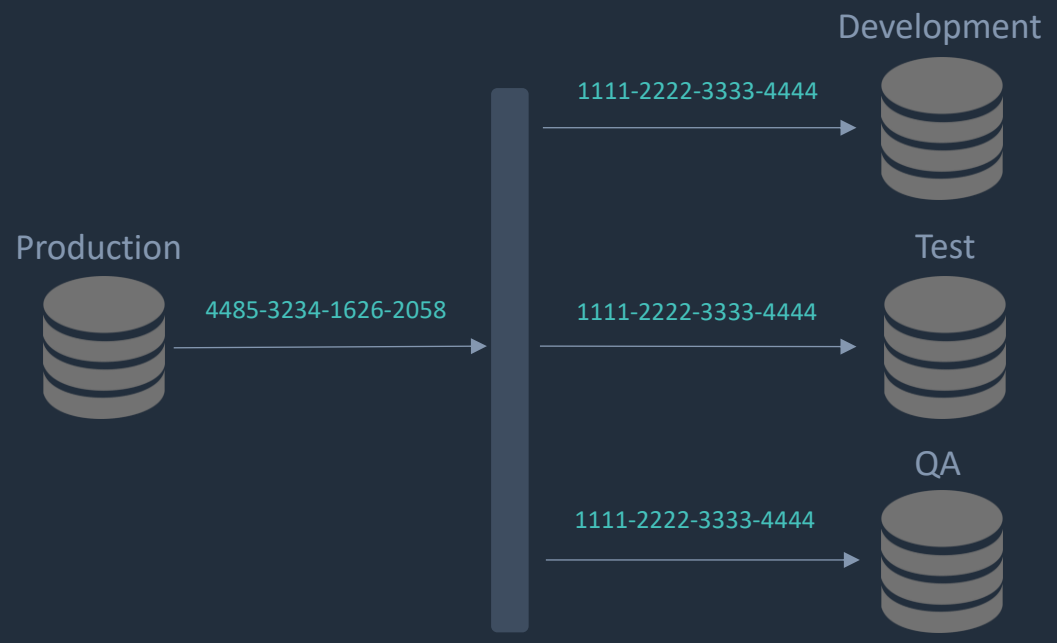
02 Techniques

Sensitive Data Obfuscation – Use Cases

Sensitive data obfuscation is a technique used to conceal sensitive information while maintaining its format and usability for authorized users and processes.

Use cases

- Compliance with regulations (GDPR, HIPAA, PCI DSS)
- Testing and development
- Data analytics and business intelligence
- Outsourcing and offshoring
- Training and education



Sensitive Data Obfuscation Techniques

Masking And Anonymization

Masking

Alters sensitive information while retaining its original structure to prevent disclosure

customer_id	card_number
1	1111-2222-3333-4444
2	5555-6666-7777-8888

customer_id	card_number
1	****-****-****-4444
2	****-****-****-8888

Anonymization

Removes (or replaces) user-identifiable information from the data so that it cannot be linked back to a user

customer_id	order_id	order_total
1	23	4000
2	27	3000

order_id	order_total
23	4000
27	3000

Pseudonymization And Aggregation

Pseudonymization

Replaces user-identifiable information with some ID. The data can be linked back to a user if some additional information is provided

customer_id	order_id	order_total
1	23	4000
2	27	3000

Hash
or
Encryption

customer_id	order_id	order_total
5B60F221...	23	4000
F3457DAB...	27	3000

Aggregation

Combines or summarizes multiple rows of data into a single result or value

customer_id	order_id	product_id	order_total
1	23	PRD0001	8000
2	27	PRD0100	4000
...	...	...	...
10 000	74516	PRD0001	12000

purchase_count	product_id	order_total
342	PRD0001	470000
117	PRD0002	350000
...	...	...
56	PRD0100	72000

Shuffling

Anonymizes or obfuscates sensitive data by randomly reordering the values within a dataset while preserving certain characteristics or relationships

customer_id	order_id	order_total
1	23	4000
2	27	3000

customer_id	order_id	order_total
2	23	4000
1	27	3000

Precalculated Tables And Indexed Views

Precalculated Tables And Indexed Views – Section Overview

01 Numbers Table

02 Calendar Table

03 Indexed Views

Numbers Table

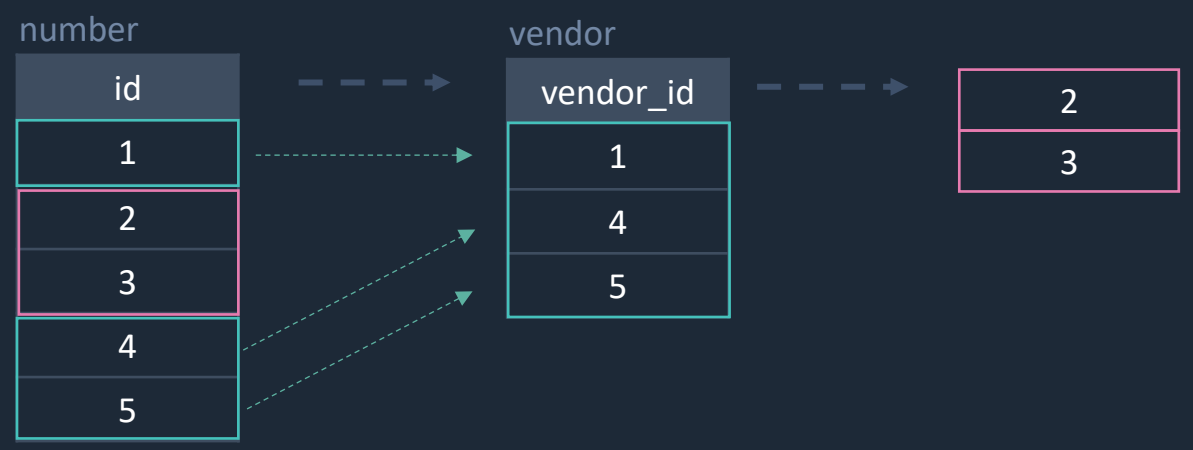
A numbers table is a precalculated table containing a sequential list of numbers

Use cases

- Sequence generation
- Missing values identification
- String manipulation

date generation example

finding missing vendor ids example



Calendar Table

A calendar table is a table that contains information about dates, such as year, month, working days, and holidays

Use cases

- Date-based analysis
- Date range filtering
- Date calculations

calendar

date_value	next_date	year	quarter	month	week	day_of_week_long_name	is_weekend
2023-06-17	2023-06-18	2023	2	6	24	Saturday	1
2023-06-18	2023-06-19	2023	2	6	25	Sunday	1
2023-06-19	2023-06-20	2023	2	6	25	Monday	0

calculation simplification example

```
…
(
   CASE t1.date_value
     WHEN DATEFROMPARTS(YEAR(t1.date_value), 12, 25) THEN 1
     WHEN DATEFROMPARTS(YEAR(t1.date_value), 1,  1) THEN 1
   ELSE 0 END
) AS 'is_public_holiday’
…
```

--------▶ calendar.is_public_holiday

Indexed View

An indexed view is a pre-computed data set created by combining data from multiple existing tables

Benefits

- Improved query performance

Drawbacks

- Limited usage (no subqueries, left, right joins, window-functions)
- DML query performance degradation
- Increased disk space usage

Statistics

Statistics in SQL Server

Statistics are metadata about the distribution of values in one or more columns of a table or indexed view

Characteristics

- Statistics enable the query optimizer to generate efficient execution plans
- Two types of statistics
  - Index statistics
  - Column statistics
- Can be created automatically (AUTO CREATE STATISTICS) or manually (CREATE STATISTICS)
- Require maintenance
  - Automatic updates (AUTO UPDATE STATISTICS)
  - Manual updates (UPDATE STATISTICS)
  - Combined approach