Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

ВВЕДЕНИЕ В АНАЛИЗ БОЛЬШИХ ДАННЫХ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения

Гомель 2023

УДК 004.04(075.8)
ББК 22.19я73
В24

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол №4 от 06.12.2021 г.*)

Составитель *Е. В. Комракова*

Рецензент: зав. каф. «Информатика» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Т. А. Трохова*

В24 **Введение** в анализ больших данных : учеб.-метод. пособие для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» днев. и заоч. форм обучения / сост. Е. В. Комракова. – Гомель : ГГТУ им. П. О. Сухого, 2023. – 129 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Предназначено для теоретической и практической подготовки по дисциплине «Введение в анализ больших данных». Знакомит студентов с основами теории и практики, необходимой для анализа больших объемов данных при исследовании реальных объектов.

Для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)» и 1-40 80 04 «Информатика и технологии программирования» дневной и заочной форм обучения.

УДК 004.04(075.8)
ББК 22.19я73

ВВЕДЕНИЕ

Учебно-методическое пособие, предназначенное для студентов специальности 1-40 05 01-01 «Информационные системы и технологии (в проектировании и производстве)». В пособии последовательно излагается материал, посвященный теории и практике, необходимыми для анализа больших объемов данных при исследовании реальных объектов.

В первом разделе описана работа с SQL Workbench и различными операторами, такими как: SELECT, DML (язык манипулирования данными), TCL (язык управления транзакциями), DDL (язык определения данных), DCL (язык управления данными). Также дается введение в OLAP и оконные функции.

Второй раздел посвящен DWH и качеству данных. Рассмотрены основы DWH, качество данных для приема и тестирования миграции, параметры качества данных, тестовые среды. Также даются подходы к работе с неструктурированными данными и работа с базами данных NoSQL.

Основные понятия анализа данных

Данные – это воспринимаемые человеком факты, события, сообщения, измеряемые характеристики, регистрируемые сигналы.

Специфика данных в том, что они, с одной стороны, существуют независимо от наблюдателя, а с другой – становятся собственно «данными» лишь тогда, когда существует целенаправленно собирающий их субъект. В итоге: данные должны быть тем основанием, на котором возводятся все заключения, выводы и решения. Они вторичны по отношению к цели исследования и предметной области, но первичны по отношению к методам их обработки и анализа, извлекающим из данных только ту информацию, которая потенциально доступна в рамках отобранного материала.

Анализ данных – это совокупность методов и средств извлечения из организованных данных информации для принятия решений.

Этапы решения задачи анализа данных и их взаимосвязи

Постановка задачи (является определяющим этапом, от которого зависит весь ход анализа) начинается со стадии формулировки цели всего исследования, ради достижения которой и предпринимаются сбор и обработка данных. Исходя из цели, определяется состав данных, которые необходимо собрать. Одна из типичных ошибок исследователей состоит в том, что сначала собираются данные, а затем начинают формулироваться задачи их обработки. Заранее собранные данные могут отражать совсем другие характеристики явления, нежели те, которые важны для поставленной цели.

Типичная форма при сборе данных – таблица «объект – признак», в которую заносятся значения признаков (свойств), характеризующие каждый исследуемый объект. Примерами признаков могут быть «вес», «длина», «цвет», «профессия», «пол», наличие или отсутствие симптома и т. д. Объекты – «люди», «изделия», «услуги» и т. д. Таблицей такого вида принято называть таблицей экспериментальных данных (ТЭД).

Состав данных – это состав признаков, которые характеризуют объекты. На стадии выбора средств происходит выбор пакета программ или системы анализа данных. Факторы, влияющие на выбор средств: объема данных, числа объектов и признаков, типов признаков, типов доступных ЭВМ, квалификации пользователя.

На стадии формализация собранных данных ТЭД необходимо придать такой вид, какого требует от входных данных выбранная пользователем автоматизированная система анализа данных. Результатом является формализованная ТЭД, готовая к вводу в систему.

Суть второго этапа анализа заключается в том, что данные сначала вводятся в ЭВМ, где они попадают в архив данных, а затем все или некоторая часть данных выбирается из архива, после чего только начинается (но уже за пределами второго этапа) процесс, традиционно называемый обработкой.

В архиве данных при помощи специальных программ – редакторов происходит проверка введенных данных и исправление ошибок. В задании обработки указываются размеры ТЭД, место хранения данных, типы признаков в ТЭД, тип решаемой задачи, режим печати результатов и пр.

Анализ данных на качественном уровне – это попытка представить собранные данные в визуальной форме, с целью: увидеть их пригодность для проверки выдвинутых гипотез или достижения поставленной цели.

Представление данных на числовой оси называют проекцией данных на признак. Этот же признак можно изобразить, разбив всю область его значений на некоторое количество интервалов, представляющих собой гистограммой объектов по признаку.

Основные задачи при качественном анализе:

1. Экономичное, или информативное, описание данных. Содержательная постановка задачи: найти небольшое число наиболее важных свойств (характеристик, особенностей) исследуемого явления. Формальная постановка задачи – устранить дублирующие друг друга признаки или найти (построить) новые признаки (меньшее число), описывающие данные. Пример: нахождение системы признаков «размеров» – «рост» – «полнота», описывающей фигуру человека и определяющей типоразмеры при изготовлении готовой одежды.

2. Группировка (классификация) объектов. Содержательная постановка: среди множества исследуемых объектов найти группы с похожими свойствами. Формальная постановка: обнаружить в пространстве описания компактные распределения точек. Пример: классификация растений, животных по видам.

3. Исследование зависимости одного признака от остальных (описание целевого признака). Содержательная постановка: описать

взаимосвязь (зависимость) избранного свойства исследуемых объектов от остальных свойств. Функциональная постановка: найти функциональную зависимость, приближенно описывающую изменение целевого признака при изменении других признаков.

4. Распознавание образов (классификация с обучением). Содержательная постановка: найти правило, пользуясь которым можно определить принадлежность любого объекта к одному из выданных образов (классов объекта). Функциональная постановка: найти в пространстве описания область, разделяющую группу точек, соответствующих различным образам, и описать ее как функцию исходных признаков; найти к какой группе точек (образу) относятся заданные объекты.

Таким образом, на этапе качественного анализа объектом исследования является структура данных, а результатом, – как правило, информация о классе моделей, которыми можно описать явление.

На этапе количественного описания данных ведется поиск параметров моделей, созданных на предыдущем этапе. Сопоставительный анализ помогает отбирать лучшие варианты, имеющие право на существование не только как формальные результаты экспериментирования, но и как содержательно значимая информация о предметной области.

То есть происходит описание созданной модели на языках формулы, отражаются количественные характеристики анализируемых данных. Причем очень часто возникает необходимость возврата к более ранним этапам обработки и повторения всего цикла исследования.

На этапе интерпретации результатов и принятия решения принимается решение об итогах анализа данных:

– прекращение дальнейшей обработки, т.к. поставленные ранее цели достигнуты;

– решение о продолжении обработки данных с использованием других методов, возможно, с коррекцией данных;

– решение о недостаточности данных или о том, что данные не содержат достаточной информации об исследуемом явлении. В этом случае анализ начинается заново.

Таким образом: успех анализа данных зависит не столько от доступных методов, алгоритмов и систем обработки, сколько от овладения пользователем методологией их применения.

1 SQL Workbench

1.1 Оператор SELECT

Что такое язык SQL и оператор SELECT

SQL – это язык запросов, который служит для манипуляции (управления) данными в реляционных базах данных. Имеет широкую популярность и поэтому любой уважающий себя IT-к должен знать основы этого языка, так как базы данных есть практически в каждой компании.

SELECT – оператор языка SQL, относится к группе операторов манипуляции данными (Data Manipulation Language, DML) и служит для выборки данных из базы данных.

Структура sql-запросов

Общая структура запроса выглядит следующим образом:

SELECT ('столбцы или * для выбора всех столбцов; обязательно')

FROM ('таблица; обязательно')

WHERE ('условие/фильтрация, например, city = 'Moscow'; необязательно')

GROUP BY ('столбец, по которому хотим сгруппировать данные; необязательно')

HAVING ('условие/фильтрация на уровне сгруппированных данных; необязательно')

ORDER BY ('столбец, по которому хотим отсортировать вывод; необязательно')

Вот самый простой пример использования оператора SELECT.

SELECT * FROM Table

где

* – показать все данные;

FROM – из источника;

Table – название источника (в данном случае таблица).

Но, на практике, зачастую нужны не все данные из таблицы, а иногда только некоторые колонки, для этого просто указывается вместо * название нужной колонки (или колонок), например:

```
SELECT Price FROM Table
```

где, Price и есть название колонки.

Если нужно указать несколько колонок, то они просто перечисляются через запятую после оператора SELECT, например

```
SELECT price, name, model
FROM Table
```

где, price, name, model это колонки из таблицы Table.

Условие выборки – оператор WHERE

В процессе выборке достаточно часто требуется отфильтровать данные по определенному условию, т.е. не все данные, а только те, которые соответствуют условию, в конструкции SELECT для этого используется оператор WHERE.

```
SELECT price FROM Table
WHERE price > 100
```

где, WHERE и есть условие, т.е. отображаются только те строки, которые соответствуют условию (цена больше 100).

Операторы сравнения в SQL

«>» – больше чего-либо;

«<» – меньше чего-нибудь;

«=» – равно;

«<>» – не равно;

«>=» – больше или равно;

«<=» – меньше или равно.

Также можно указывать в условии ключевое слово BETWEEN, т. е. попадает или не попадает значение в определенный промежуток, например

```
SELECT price
FROM table
WHERE price BETWEEN 400 AND 600
```

где, указывается, что цена должна быть в промежутке от 400 до 600.

Для проверки, входит ли значение проверяемого выражения в какой-то определенный набор значений, можно использовать предикат IN.

```
SELECT price
FROM table
WHERE price IN (400, 600)
```

В данном случае получаем только цену со стоимостью 400 и 600.

Если нужно получить только уникальные строки источника, можно указать ключевое слово DISTINCT, например

```
SELECT DISTINCT price
FROM Table
WHERE price > 100
```

Сортировка ORDER BY

Очень часто необходимо отсортировать результат запроса по определенному полю (колонке). Для этого после запроса указывается конструкцию ORDER BY и те поля (можно несколько через запятую), по которым необходимо выполнить сортировку.

```
SELECT price
FROM Table
ORDER BY price DESC
```

Этот пример сортировки по убыванию, но можно сортировать и по возрастанию, вместо DESC пишем ASC, но обычно так не пишут, так как сортировка по возрастанию является по умолчанию.

Агрегирующие функции в SQL

В SQL очень полезные так называемые агрегирующие функции, а именно:

COUNT – количество значений в указанном столбце;

SUM – сумма значений в указанном столбце;

AVG – среднее значение в указанном столбце;

MIN – минимальное значение в указанном столбце;

MAX – максимальное значение в указанном столбце.

Например, нужно получить среднюю цену компьютера, максимальную и минимальную, для этого напишем следующий запрос

```
SELECT AVG(price), MAX(price), MIN(price)
FROM table
```

Группировка GROUP BY

Можно проводить группировку значений по колонкам, например, нужно узнать среднюю цену для каждой модели компьютера, в данном случае запрос будет вот таким

```
SELECT model, AVG (price) AS AVGPrice
```

```
FROM table
GROUP BY model
```

Также как и при использовании условий в отдельных колонках можно указывать и условие на целую группу, с помощью функции HAVING. Например, нужно определить максимальную цену компьютера, сгруппированную по моделям этих компов, но максимальная цена которых, меньше 500.

```
SELECT model, MAX(price)
FROM table
GROUP BY model
HAVING MAX(price) < 500
```

В этом случае запрос выдаст сгруппированные по моделям компьютеры, максимальная цена которых, меньше 500.

GROUP BY – необязательный элемент запроса, с помощью которого можно задать агрегацию по нужному столбцу (например, если нужно узнать какое количество клиентов живет в каждом из городов).

При использовании GROUP BY обязательно:

- перечень столбцов, по которым делается разрез, был одинаковым внутри SELECT и внутри GROUP BY,
- агрегатные функции (SUM, AVG, COUNT, MAX, MIN) должны быть также указаны внутри SELECT с указанием столбца, к которому такая функция применяется.

Группировка количества клиентов по городу:

```
select City, count(CustomerID) from Customers
GROUP BY City
```

Группировка количества клиентов по стране и городу:

```
select Country, City, count(CustomerID) from Customers
GROUP BY Country, City
```

Группировка продаж по ID товара с разными агрегатными функциями: количество заказов с данным товаром и количество проданных штук товара:

```
select ProductID, COUNT(OrderID), SUM(Quantity) from OrderDetails
GROUP BY ProductID
```

Группировка продаж с фильтрацией исходной таблицы. В данном случае на выходе будет таблица с количеством клиентов по городам Германии:

```
select City, count(CustomerID) from Customers
WHERE Country = 'Germany'
GROUP BY City
```

Переименование столбца с агрегацией с помощью оператора AS.

По умолчанию название столбца с агрегацией равно примененной агрегатной функции, что далее может быть не очень удобно для восприятия.

```
select City, count(CustomerID) AS Number_of_clients from Customers
group by City
```

NULL значение в SQL

В SQL есть такое значение как NULL. На самом деле NULL это отсутствие значения (т. е. пусто). Для того чтобы вывести все строки, в которых есть такое значение (например, у нас для какого-нибудь компьютера еще не назначена цена) можно использовать следующее условие.

```
SELECT *
FROM table
WHERE price IS NULL
```

Что и будет означать поиск всех строк, в которых отсутствует значение для поля price.

HAVING

HAVING – необязательный элемент запроса, который отвечает за фильтрацию на уровне сгруппированных данных (по сути, WHERE, но только на уровень выше).

Фильтрация агрегированной таблицы с количеством клиентов по городам, в данном случае оставляем в выгрузке только те города, в которых не менее 5 клиентов:

```
select City, count(CustomerID) from Customers
group by City
HAVING count(CustomerID) >= 5
```

В случае с переименованным столбцом внутри HAVING можно указать как и саму агрегирующую конструкцию count(CustomerID), так и новое название столбца number_of_clients:

```
select City, count(CustomerID) as number_of_clients from Customers
group by City
HAVING number_of_clients >= 5
```

Пример запроса, содержащего WHERE и HAVING. В данном запросе сначала фильтруется исходная таблица по пользователям, рассчитывается количество клиентов по городам и остаются только те города, где количество клиентов не менее 5:

```
select City, count(CustomerID) as number_of_clients from Customers
WHERE CustomerName not in ('Around the Horn','Drachenblut Delikatessend')
group by City
HAVING number_of_clients >= 5
```

JOIN

JOIN – необязательный элемент, используется для объединения таблиц по ключу, который присутствует в обеих таблицах. Перед ключом ставится оператор ON.

Запрос, в котором соединяем таблицы Order и Customer по ключу CustomerID, при этом перед названиям столбца ключа добавляется название таблицы через точку:

```
select * from Orders
JOIN Customers ON Orders.CustomerID = Customers.CustomerID
```

Нередко может возникать ситуация, когда надо промэппить одну таблицу значениями из другой. В зависимости от задачи, могут использоваться разные типы присоединений. INNER JOIN – пересечение, RIGHT/LEFT JOIN для мэппинга одной таблицы знаениями из другой,

```
select * from Orders
join Customers on Orders.CustomerID = Customers.CustomerID
where Customers.CustomerID >10
```

Другие типы JOIN'ов можно увидеть на рисунке 1.1.

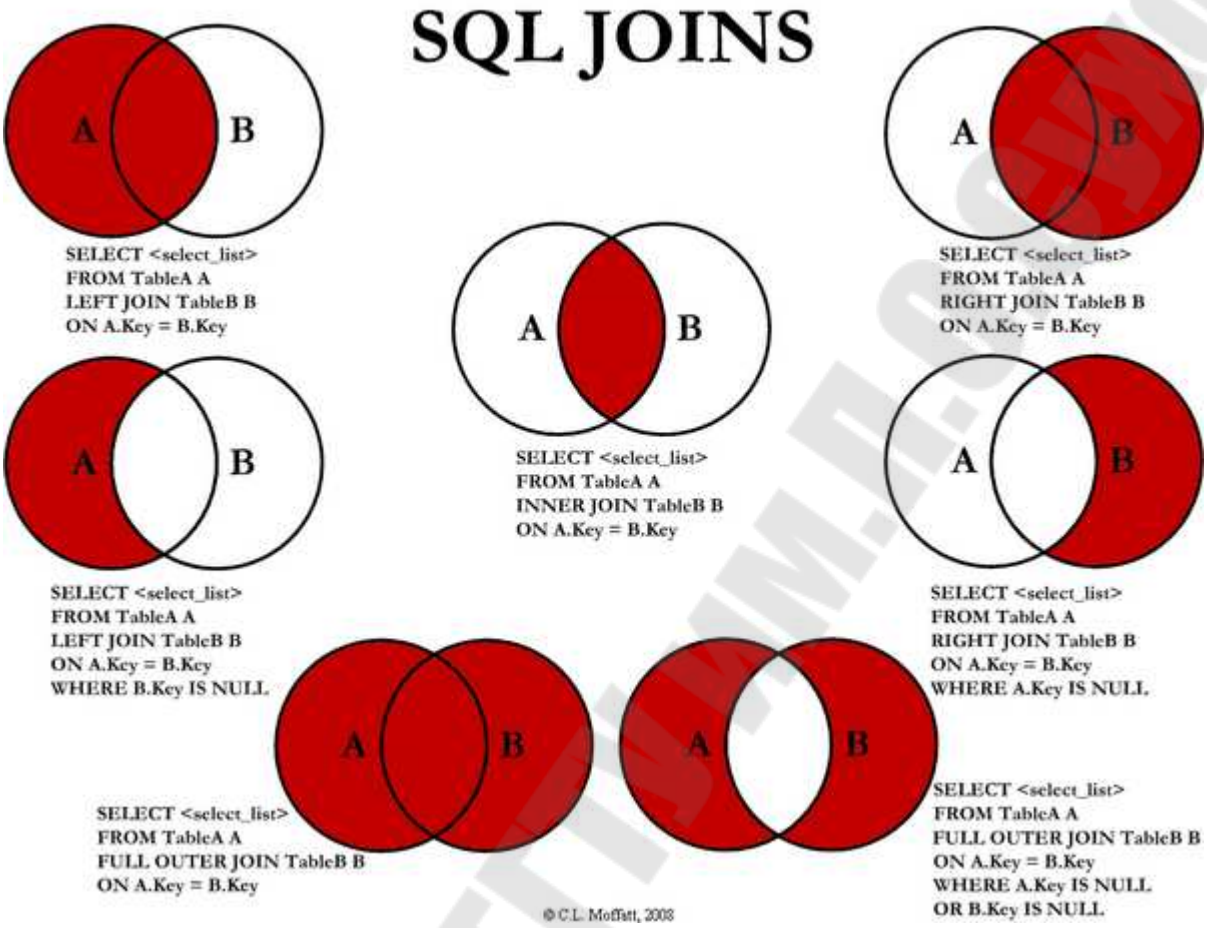

Рис. 1.1. – Типы запросов join

1.2 Операторы DML (язык манипулирования данными)

Данные в реляционных базах данных управляются с помощью DML (Data Manipulation Language) комманд. Эти команды это INSERT, UPDATE, DELETE и (в последних версиях SQL) MERGE.

Строго говоря существует пять DML команд

– SELECT
– INSERT
– UPDATE
– DELETE
– MERGE

На практике профессионалы в области баз данных SELECT обычно не рассматривают как часть DML.

Команда INSERT

В базе данных данные хранятся в виде строк в таблицах. Таблица наполняется строками несколькими способами, но самый частый используемый метод это команда INSERT. SQL – это язык ориантированные на работу с наборами данных, и таким образом одна команда может вилять на одну строку либо на набор строк. Отсюда следует что команда INSERT может добавить одну строку в одну таблицу или много строк в много таблиц. Базовая версия запроса добавляет всего одну строку, но сложные запросы могут добавлять несколько строк в несколько таблиц.

Простейшая форма команды INSERT добавляет одну строку в таблицу используя значения указанные в команде. Синтаксис такого запроса

INSERT INTO table [(column [,column…])] VALUES (value [,value…]);

Примеры

insert into hr.regions values (10,'Great Britain');

insert into hr.regions (region_name, region_id) values ('Australasia',11);

insert into hr.regions (region_id) values (12);

insert into hr.regions values (13,null);

Первая из команд указывает значения для обоих столбцов таблицы REGIONS. Если у таблицы есть третий столбец то запрос выполнится неуспешно так как команда использует позиционное обозначение (positional notation). В команде не указывается в какой столбец необходимо вставить конкретное значение, запрос рассматривает позицию значений, их порядок в команде. Когда БД получает запрос использущий позиционное обозначение она будет сопостовлять порядок значений со порядком определения столбцов при создании. Запрос выполнится неуспешно если порядок будет неверный: БД попробует вставить данные, но типы данных столбцов разные.

Второй запрос указывает и столбцы и значения которые использовать для столбцов. Обратите внимание что теперь порядок определения столбцов в таблице не важен – важен порядок столбцов и значений в запросе.

Третий пример указывает один столбец и одно значение. Для всех остальных столцов будет использоваться значение NULL. Запрос выполнится неуспешно если столбец REGION_NAME обязательный (not null). Четвертый пример приведет к такому же результату как и третий, но так как не были указаны столбцы в запросе – необходимо указать значения (даже NULL) явно для всех столбцов.

Для вставки нескольких строк одним запросом значения для строк должны возвращаться запросом. Синтаксис такой команды

INSERT INTO table [column [, column…]] subquery;INSERT INTO table [column [, column…]] subquery;

Обратите внимание что такой синтаксис не использует ключевое слово VALUES. Если список столбцов пропущен то подзапрос должен возвращаться значения для каждого столбца таблицы. Для копирование всех строк из одной таблицы в другую если у таблиц одинаковые столбцы то команда для такой операции будет вида

insert into regions_copy select * from regions;

Такой запрос предполагает что таблица regions_copy уже существует. Подзапрос SELECT считывает все строки из таблицы-источника (REGIONS) и команда INSERT записывает все строки в таблицу-цель (REGIONS_COPY)

Команда UPDATE

Команда UPDATE используется для изменения строк которые уже существуют – строки которые были созданы с помощью команды INSERT или возможно другими инструментами такими как Data Pump. Как и другие SQL команды, команда UPDATE может влиять на одну строку или набор строк. Размер набора данных обновляемым командой UPDATE определяется условием WHERE, точно таким же образом как и набор строк получаемый командой SELECT. Синтаксис идентичный. Все обновляемые строки будут находиться в одной таблице; невозможно одной командой UPDATE обновлить данные в нескольких таблицах.

Когда обновляются данные команда UPDATE указывает какие столбцы набора строк обновлять. Необязательно обновлять все столбцы строки. Если обновляемые столбец уже хранит значение, оно будет заменено на новое указанное в команде UPDATE. Если в столбец не было значения – т.е. было значение NULL – то столбец будет обновлен на новое значение.

Обычное использование UPDATE – это получение одной строки и обновление одного или нескольких столбцов в этой строке. Строка получается используя условие WHERE по первичному ключу, уникальному идентификатору которые гарантирует что только одна строка будет получена. Затем обновляются столбцы которые не являются столбцами первичного ключа. Обычно значение первичного ключа не изменяется. Жизненный цикл строки начинается когда она добавляется, затем может происходить несколько изменений до тех пор пока строка не удаляется и жизненный цикл не заканчивается. Во время жизни строки обычно первичный ключ не изменяется.

Команда UPDATE должна соблюдать все ограничения наложенные на таблицу, так же как и команда INSERT. Например невозможно обновить значение столбца с ограничением обязательности (not null) на значение NULL или обновить первичный ключ на неуникальное значение. Базовый синтаксис команды UPDATE

UPDATE table SET column=value [,column=value…] [WHERE condition];

Синтаксис для команды UPDATE использующей подзапросы

```
UPDATE table
SET column=[subquery] [,column=subquery…]
WHERE column = (subquery) [AND column=subquery…] ;
```

Существует жесткое ограничение на результат возвращаемый подзапросом для определения значения: подзапрос должен возвращать скалярное значение. Скалярное значение это одно значение (одна строка, один столбец) необходимого типа данных. Если подзапрос вернет не скалярное значение запрос не выполнится. Рассмотрим два примера

```
update employees
set salary=(select salary from employees where employee_id=206);
update employees
set salary=(select salary from employees where last_name='Abel');
```

Первый пример использует предикат равенства по первичному ключу и запрос всегда выполнится успешно. Даже если запрос не вернет ни одну строку (если нет сотрудника с номером 206) запрос вернет скалярное значение: NULL. В этом случае у всех сотрудников зарплата станет NULL – что может быть не совсем верно с точки зре-

ния качества данных, но с точки зрения SQL здесь нет ошибки. Второй запрос использует предикат равенства для поля LAST_NAME, что не гарантирует уникальность. Запрос выполнится успешно если существует только один сотрудник с таким именем, но если в таблице больше чем одна строка с таким значение запрос вернет ошибку “ORA-01427: single-row subquery returns more than one row.” Для надежного кода неважно состоянии данных, важно убедиться что подзапросы всегда возвращают скалярное значение.

Подзапросы в условии WHERE тоже должны возвращать скалярное значение если используется предикат равенства или предикат отношения > или <. Если используется предикат вхождения (IN) то подзапрос может возвращать набор строк. К примерму

update employees
set salary=10000
where department_id in (select department_id from departments where department_name like ‘%IT%’);

Результатом этого запроса будет обновление значения зарплаты всем сотрудникам название отдела которых содержит подстроку IT. Но несмотря на то что подзапрос может возвращать в таких случаях несколько строк – он все равно должен возвращать один столбец.

Команда DELETE

Ранее добавленные строки можно удалить из таблицы используя команду DELETE. Эта команда удалит одну или несколько строк из таблицы в зависимости от условия в секции WHERE. Если условие WHERE пропущено то все строки будут удалены из таблицы.

Удаление столбцов происходит по принципу либо все либо ничего. Нельзя указать столбец в команде DELETE. Когда строка добавляется в таблицу нужно указать столбцы для заполнения. Когда строка обновляется можно выбрать столбцы для обновления. Но удаление происходит для всей строки – единственным выбором является какие строки удалять. Это делает команду DELETE легче чем другие команды с точки зрения синтаксиса. Синтаксис команды DELETE

DELETE FROM table [WHERE condition];

Это простейшая команда DML, особенно если условие WHERE будет пропущено. В этом случае все строки будут удалены. Единственным усложнением команды может быть добавление условия. К примеру условия равенства/подобия литералу

```
delete from employees where employee_id=206;
delete from employees where last_name like 'S%';
delete from employees where department_id=&Which_department;
delete from employees where department_id is null;
```

Первый запрос идентифицирует строку по первичному ключу. Одна строка будет удалена – или одна или ни одной, если заданное значение ключа не найдено в таблице. Второй запрос использует предикат подобия что может привести к удалению многих строк: будут удалены все сотрудники фамилия которых начинается с буквы S. Третий запрос запросит вводи значения для переменной в запросе и все сотрудники департамента будут удалены. Последний запрос удалит всех сотрудников у которых не назначен департамент (значение департамента NULL).

Если команда DELETE не удаляет ни одной строки – это не рассматривается как ошибка. Команда вернет сообщение "0 rows deleted' вместо сообщения об ошибки посколько команды выполнена успешна – просто не были найдены строки для удаления.

Команда MERGE

Часто возникает ситуация когда необходимо взять набор данных (источник) и интегрировать его в существующую таблицу (цель). Если строка источника уже существует в таблице-цели, то возможно обновить строку в таблице-цели, или удалить старую строку и вставить новую или можно вообще не трогать такие строки. Если строка источника не существует в таблице-цели, то можно добавить такую строку. Команда MERGE позволяет сделать это. MERGE работает с наборами данных, для каждой строки источника пытается найти уже существующую строку в таблице-цели. Если совпадение не найдено – строка будет добавлена; если строка найдена то она может быть обновлена.

Команда MERGE не делает ничего такого что было бы невозможно сделать командами DELETE, INSERT и UPDATE – но только она может сделать это за один проход данных. Альтернативой команде MERGE будет три прохода данных, по одному для каждой команды.

Источником для команды MERGE может быть таблица или запрос. Условие для нахождения совпадения аналогично условию WHERE. Секция которая отвечает за обновление или добавление данных аналогично соответствующей команде INSERT или UPDATE.

Получается что команда MERGE самая сложная из DML команд (сложно не согласиться) и самая мощная (что можно оспорить). Рассмотрим простой пример

```
merge into employees e
using new_employees n on (e.employee_id = n.employee_id)
when matched then
update set e.salary=n.salary
when not matched then
insert (employee_id,last_name,salary)
values (n.employee_id,n.last_name,n.salary);
```

Данный запрос использует таблицу NEW_EMPLOYEES для обновления или добавления данных в таблицу EMPLOYEES. Команда пройдет по данным таблицы NEW_EMPLOYEES и для каждой строки этой таблицы попробует найти строку в таблице EMPLOYEES соответствующую заданному условию. Если строка найдена значение поля SALARY будет обновлено на новое значение из таблицы NEW_EMPLOYEES. Если строка не найдена одна новая строка будет добавлена.

Неуспешное выполнение DML команд

Запрос может выполниться неуспешно по многим причинам, включая следующие

1. Ошибка в синтаксисе
2. Ссылка на несуществующий объект или столбец
3. Недостаток прав
4. Нарушение ограничений
5. Проблемы с доступным местом

1.3 Операторы TCL (язык управления транзакциями)

Обеспечение функционирования баз данных

Восстановление СУБД означает восстановление самой базы данных, т. е. возвращение БД в правильное состояние. Основной принцип, на котором строится такое восстановление – это избыточность, которая организуется на физическом уровне.

Транзакцией называется последовательность операций, производимых над базой данных, переводящая базу данных из одного непротиворечивого состояния в другое непротиворечивое состояние.

Стандартные команды для работы с транзакциями:

BEGIN TRAN – начало транзакции,

ROLLBACK TRAN – откат, отмена транзакции; все изменения, сделанные с начала транзакции, будут отменены,

COMMIT TRAN – завершение, подтверждение транзакции; все изменения, сделанные с начала транзакции, будут зафиксированы .

До момента подтверждения транзакции все измененные данные записываются в журнал транзакций, и только после фиксации транзакции данные переносятся собственно в таблицы .

Уровни изолированности транзакций:

Serializable – самый надежный, но и самый медленный уровень изолированности, транзакции выполняются последовательно, друг за другом.

Repeatable read – при повторном чтении данных результат будет точно таким же, как и при первом чтении, даже если данные были изменены.

Read commited – допускается читать только данные завершенных транзакций.

Read uncommited - допускается читать “грязные данные”, т. е., данные незавершенных транзакций.

Рассмотрим пример :

Предположим, что отношение P (отношение деталей) включает атрибут TOTQTY, представляющий собой общий объем поставок для каждой детали. Значение TOTQTY для любой определенной детали предполагается равным сумме всех значений QTY для всех поставок данной детали. Ниже показано добавление в базу данных новой поставки со значением 1000 для поставщика S5 и детали P1.

```
BEGIN TRANSACTION;
INSERT ({S#:’S5’, P#:’P1’, QTY:1000}) INTO SP;
IF ошибка THEN GO TO UNDO;
UPDATE P WHERE P# = ’P1’ TOTQTY:=TOTQTY+1000;
IF ошибка THEN GO TO UNDO;
COMMIT TRANSACTION;
GO TO FINISH;
UNDO: ROLLBACK TRANSACTION;
FINISH: RETURN;
```

Восстановление транзакции

Транзакция начинается с успешного выполнения оператора Begin Transaction и заканчивается успешным выполнение либо оператора Commit, либо оператора Rollback. Оператор Commit устанавливает так называемую точку фиксации, которая соответствует концу логической единице работы и , следовательно, точке, в которой база данных находится в согласованном состоянии. Выполнение же оператора Rollback возвращает базу данных в состоянии, в котором она находилась во время выполнения оператора Begin Transaction, т.е. в предыдущую точку фиксации.

Из этого следует, что транзакция – это не только логическая единица работы, но и также единица восстановления при неудачном выполнении операций.

Транзакции обладают четырьмя важными свойствами : атомарность, согласованность, изоляция и долговечность.

1. Атомарность. Транзакции атомарны (выполняется все или ничего).

2. Согласованность. Транзакции защищают БД согласованно, т.е. транзакции переводят одно согласованное состояние БД в другое без обязательной поддержки согласованности в промежуточных точках.

3. Изоляция. Транзакции отделены друг от друга. Это означает, что при запуске множества конкурирующих друг с другом транзакций, любое обновление определенной транзакции будет скрыто от остальных до тех пор, пока эта транзакция выполняется.

4. Долговечность. Когда транзакция выполнена, ее обновления сохраняются, даже если в следующий момент произойдет сбой системы.

Управление транзакциями

Для управления транзакциями используются следующие команды:

– COMMIT – сохранить изменения.

– ROLLBACK – отменить изменения.

– SAVEPOINT – создает точки сохранения в группах транзакций.

– SET TRANSACTION – помещает имя в транзакцию.

Команды управления транзакциями

Команды управления транзакциями используются только с командами DML, такими как – INSERT, UPDATE и DELETE. Они не могут использоваться при создании таблиц или их удалении, поскольку эти операции автоматически фиксируются в базе данных.

Команда COMMIT

Команда COMMIT – это транзакционная команда, используемая для сохранения изменений внесенных транзакцией в базу данных. Команда COMMIT сохраняет все транзакции в базе данных с момента выполнения последней команды COMMIT или ROLLBACK.

Синтаксис команды COMMIT следующий.

```
1    COMMIT;
```

Пример

Рассмотрим таблицу CUSTOMERS, содержащую следующие записи:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Рис. 1.2. Таблица CUSTOMERS

Ниже приведен пример, в котором из таблицы будут удалены клиенты с возрастом = 25, а затем эти изменения будут сохранены в базе данных.

```
1    DELETE FROM CUSTOMERS
2      WHERE AGE = 25;
3    COMMIT;
```

Таким образом, из таблицы будут удалены две строки, и результат выполнения инструкции SELECT будет выглядеть следующим образом.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Рис. 1.3. Таблица CUSTOMERS после выполнения инструкии SELECT

Команда ROLLBACK

Команда ROLLBACK – это транзакционная команда, используемая для отмены транзакций, которые еще не были сохранены в базе данных. Эта команда может использоваться только для отмены транзакций с момента выполнения последней команды COMMIT или ROLLBACK.

Синтаксис команды ROLLBACK следующий:

```
1    ROLLBACK;
```

Ниже приведен пример, в котором из базы данных будут удалены все записи для которых возраст = 25, а затем эти изменения будут отменены.

```
1    DELETE FROM CUSTOMERS
2      WHERE AGE = 25;
3    ROLLBACK;
```

Таким образом, операция удаления не приведет к изменениям в таблице, и результат выполнения инструкции SELECT будет выглядеть следующим образом.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Рис. 1.4. Таблица CUSTOMERS после выполнения команды ROLLBACK

Команда SAVEPOINT

SAVEPOINT – это точка транзакции, к которой можно вернуть транзакцию, не откатывая ее полностью. Синтаксис команды SAVEPOINT приведен ниже.

```
1       SAVEPOINT SAVEPOINT_NAME;
```

Эта команда предназначена только для создания SAVEPOINT в других транзакционных операторах. Команда ROLLBACK используется для отмены группы транзакций до точки SAVEPOINT.

Синтаксис, который используется для возврата к SAVEPOINT, показан ниже.

```
1       ROLLBACK TO SAVEPOINT_NAME;
```

Ниже приведен пример ситуации, когда планируется удалить три разных записи из таблицы CUSTOMERS. Можно создать SAVEPOINT перед каждым удалением, чтобы иметь возможность откатить операцию к любой SAVEPOINT любое время и вернуть данные в исходное состояние.

Пример

Следующий блок кода содержит ряд операций.

```
1       SAVEPOINT SP1;
2       Savepoint created.
3       DELETE FROM CUSTOMERS WHERE ID=1;
4       1 row deleted.
5       SAVEPOINT SP2;
6       Savepoint created.
```

```
7       DELETE FROM CUSTOMERS WHERE ID=2;
8       1 row deleted.
9       SAVEPOINT SP3;
1       Savepoint created.
1       DELETE FROM CUSTOMERS WHERE ID=3;
1       1 row deleted.
```

Теперь, после того как были выполнены изменения, предположим, что было решено откатить операцию к точке SAVEPOINT, которая определена как SP2. Поскольку SP2 была создана после первого удаления, последние два удаления будут отменены:

```
1       ROLLBACK TO SP2;
2       Rollback complete.
```

Обратите внимание, что перед тем как вернулись к SP2, было произведено первое удаление

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
6 rows selected.
```

Рис. 1.5. Таблица CUSTOMERS после выполнения команды ROLLBACK с использованием SAVEPOINT

Команда RELEASE SAVEPOINT

Команда RELEASE SAVEPOINT используется для удаления созданной точки SAVEPOINT. Синтаксис команды RELEASE SAVEPOINT следующий.

```
1       RELEASE SAVEPOINT SAVEPOINT_NAME;
```

После того как SAVEPOINT будет удалена, не будет возможности использовать команду ROLLBACK для отмены транзакций, выполненных после последней SAVEPOINT.

Команда SET TRANSACTION

Команда SET TRANSACTION может использоваться для инициирования транзакции базы данных. Эта команда используется для указания характеристик транзакций, которая задается после команды. Например, можно задать для транзакции режим только чтения или чтение и запись. Синтаксис команды SET TRANSACTION следующий.

```
1        SET TRANSACTION [ READ WRITE | READ ONLY ];
```

1.4 Операторы DDL (язык определения данных)

Язык SQL имеет две составляющие: язык обращения с данными Data Manipulation Language (DML) и язык определения данных Data Definition Language (DDL). DML состоит из операторов, используемых для создания и получения данных. DDL состоит из операторов, используемых для создания объектов в базе данных и для установки свойств и значений атрибутов самой базы данных.

DML и DDL

Чем же отличаются эти две группы операторов? В то время, как операторы DML достаточно однотипны для различных реализаций SQL (что дает возможность каждому поставщику программной продукции вводить свои расширения), DDL имеет существенные различия для разных продуктов. Каждый поставщик системы управления базой данных на физическом уровне различным образом реализует реляционную модель и каждый поставщик DDL неизбежно отражает эти различия. Большинство поставщиков предоставляют графические инструменты для определения данных и многие, включая и Microsoft, не ограничиваются использованием только SQL DDL. Например, Microsoft предоставляет поддержку двух стандартов определения данных: ADO и DAO.

Базовыми же операторами SQL DDL являются CREATE, ALTER и DROP, каждый из которых имеет несколько вариаций для создания объектов различных типов.

Операторы языка описания данных

Перечислим основные операторы языка SQL, с помощью которых создается и изменяется структура базы данных:

– CREATE TABLE – создание структуры таблицы;

– ALTER TABLE – изменение структуры таблицы;
– DROP TABLE – удаление структуры таблицы;
– CREATE VIEW – создание представления;
– DROP VIEW – удаление представления;
– CREATE INDEX – создание индекса (не входит в стандарт);
– DROP INDEX – удаление индекса;
– GRANT – предоставление привилегий (прав) пользователям;
– REVOKE – удаление привилегий;
– CREATE SCHEMA – задание последовательности операторов
– DROP SCHEMA – удаление схемы.

Часть языка SQL, в которой используются эти операторы, называют языком описания данных (DDL - Data Definition Language).

Оператор CREATE TABLE

Создание таблицы – это запоминание в базе данных структуры таблицы, описанной в операторе CREATE TABLE. Запоминание происходит в результате выполнения этого оператора. Рассмотрим простейший пример создания таблицы.

Пример. Создание таблицы из четырех столбцов.

```
CREATE TABLE knigi
( naim CHAR(100),
  kvo_str INTEGER,
  god_izd SMALLINT,
  avtor CHAR(60)
);
```

Обозначения:

knigi - имя таблицы;

naim, kvo_str, god_izd, avtor – имена полей (столбцов);

INTEGER, SMALLINT, CHAR(100),CHAR(60) - типы данных

100, 60 – длина поля в символах.

Таблица может быть временной. Описание структуры временной таблицы хранится в базе данных постоянно, а данные удаляются, когда исчезает необходимость в них. Таблица объявляется временной ключевым словом TEMPORARY. Ключевое слово GLOBAL задает удаление данных в конце сеанса пользователя, LOCAL – по завершении программы или модуля. ON COMMIT DELETE ROWS – удаление данных по завершению транзакции. Если задано ON COMMIT

PRESERVE ROWS, то данные во временной таблице по завершении транзакции сохраняются.

Ограничения столбца имеют следующие значения:

– NOT NULL – в столбце не допускается отсутствие значения;

– UNIQUE – все значения в столбце должны быть уникальными, но допускается NULL;

– PRIMARY KEY – действует так же, как UNIQUE, но NULL не допускается;

– CHECK(условие) – проверка условия

– DEFAULT = выражение – значение по умолчанию ;

– REFERENCES имя таблицы [(имя столбца)] – внешний ключ. Имя столбца указывается, когда столбец не является первичным ключом.

Ограничение может иметь имя, которое задается перед ограничением ключевым словом CONSTRAINT:

CONSTRAINT имя ограничения ,

например,

CONSTRAINT znach NOT NULL .

Ограничения таблицы:

– UNIQUE (список столбцов)) – все комбинации значения столбцов должны быть уникальными, но допускается NULL;

– PRIMARY KEY (список столбцов) действует так же, как UNIQUE, но NULL не допускается;

– CHECK (условие) – проверка условия

– FOREIGN KEY (список столбцов) имя таблицы [(список столбцов)] – внешний ключ. Список столбцов связанной таблицы указывается, когда он не является первичным ключом.

Пример создание двух связанных таблиц. Рассмотрим сущности писатель и книга. Для простоты будем считать, что у книги может быть только один автор. Тогда между сущностями писатель и книга существует связь типа один ко многим. Для отображения этой связи в реляционной модели создадим в таблице pisatel первичный ключ Id_p, а в таблицу kniga добавим внешний ключ Id_pisatel. Кроме того, зададим адрес писателя по умолчанию и проверку числа страниц в книге. Соответствующие запросы будут выглядеть так:

```
CREATE TABLE pisatel
( Id_p SMALLINT PRIMARY KEY,
```

```
  FIO CHAR(60) NOT NULL UNIQUE,
  adr CHAR(100) DEFAULT='Москва'
);

CREATE TABLE kniga
( naim CHAR(100) NOT NULL UNIQUE,
  kvo_stranic INTEGER CHECK ((kvo_stranic<10000) AND
(kvo_stranic > 0))
  god_izd SMALLINT,
  Id_pisatel SMALLINT REFERENCES pisatel
);
```

В таблице kniga поле naim = потенциальный ключ.

Пример составного первичного ключа. В таблице tovar пара значений наименование, изготовитель должна быть уникальной.

```
CREATE TABLE tovar
( наименование CHAR(50),
  изготовитель CHAR(50),
  цена MONEY,
  PRIMARY KEY (наименование,изготовитель)
);
```

Пример составного потенциального ключа. Поле Id – первичный ключ, пара полей наименование,изготовитель – потенциальный ключ.

```
CREATE TABLE товар1
( Id SMALLINT PRIMARY KEY,
  наименование CHAR(50) NOT NULL,
  изготовитель CHAR(50) NOT NULL,
  цена MONEY,
  UNIQUE (наименование,изготовитель)
);
```

Пример составного внешнего ключа. Пара полей изделие, фирма таблицы заказ – внешний ключ. Эта пара связана с потенциальным ключем наименование,изготовитель таблицы товар

```
CREATE TABLE заказ
( N_заказа INTEGER NOT NULL UNIQUE,
  изделие CHAR(50),
  фирма CHAR(60),
  к_во SMALLINT CHECK((к_во > 0) AND ,(к_во <= 100)),
```

FOREIGN KEY (изделие,фирма) товар1(наименование,изготовитель)
);

Оператор ALTER TABLE

Оператор ALTER TABLE служит для изменения структуры существующих таблиц. В реляционной базе данных существует множество часто неявных, скрытых связей, которые при изменении структуры таблиц могут быть нарушены. Поэтому применять оператор ALTER TABLE следует крайне осторожно.

Синтаксическая формула оператора ALTER TABLE:

ALTER TABLE имя таблицы
{ADD[COLUMN] {имя столбца} тип[(длина)] [ограничение ...]}
|{ALTER [COLUMN} {имя столбца} {изменяющее действие}
|{DROP [COLUMN] {имя столбца} RESTRICT | CASCADE}
|{ADD ограничение таблицы}
|{DROP CONSTRAINT имя ограничения RESTRICT | CASCADE};

изменяющее действие ::= {SET DEFAULT значение по умолчанию} | {DROP DEFAULT}

Пример добавления столбца. К таблице заказ добавляется столбец цена, имеющий тип MONEY.

ALTER TABLE заказ ADD цена MONEY;

Оператор DROP TABLE

Синтаксическая формула оператора DROP TABLE:

DROP TABLE имя таблицы;

Пример

DROP TABLE kniga;

Оператор CREATE INDEX

Синтаксическая формула оператора CREATE INDEX:

CREATE [UNIQUE] INDEX имя индекса
ON имя твблицы(имя столбца .,..);

Пример. Создается индекс kniga_ind для таблицы kniga.

CREATE INDEX kniga_ind
ON kniga(naim);

Оператор DROP INDEX

Синтаксическая формула оператора DROP INDEX:

DROP INDEX имя индекса;

Пример

DROP INDEX kniga_ind;

Оператор CREATE VIEW

В контексте реляционных баз данных термин VIEW переводится на русский язык как представление. Синтаксическая формула оператора CREATE VIEW:

CREATE VIEW имя таблицы[(столбец .,..)]
(AS оператор запроса SELECT
[WITH [CASCADED | LOCAL] CHECK OPTION]);

Оператор CREATE VIEW необычен тем, что содержит в себе оператор SELECT, принадлежащий языку манипулирования данными. Заметьте, что в синтактической формуле используется не имя представления, что было бы логично, а имя таблицы. Когда в каком-либо запросе языка манипулирования данными встречается имя таблицы, объявленное в операторе CREATE VIEW, то выполняется запрос SELECT, объявленный в том же CREATE VIEW. Результаты этого запроса рассматриваются как обыкновенная таблица!

Данные могут извлекаться в представление из одной или из нескольких настоящих таблиц. Источником формирования представления может быть другое представление. В том случае, когда представление связано только с одной таблицей, оно может использоваться для изменения (оператор UPDATE), удаления (оператор DELETE) и добавления (оператор INSERT) данных в породившую его таблицу.

Предложение CHECK OPTION служит для проверки нарушения целостности данных при использовании представления в операторах INSERT и UPDATE. CASCADED распространяет проверку на все уровни вложенности представления, а LOCAL ограничивает проверку только одним уровнем.

Пример. Создается представление, в которое отбираются только дешевые товары из таблицы товар

CREATE VIEW дешевый_товар
(AS SELECT * FROM товар
WHERE цена < 100);

Оператор DROP VIEW

Синтаксическая формула оператора удаления представления DROP VIEW:

DROP VIEW имя представления RESTRICT | CASCADE

RESTRICT вызывает сообщение об ошибке при существовании ссылки на это представление.

При задании CASCADE удаляются все объекты, в которых есть ссылки на удаляемое представление.

1.5 Операторы DCL (язык управления данными)

Data Control Language (DCL) – это группа операторов определения доступа к данным. Иными словами, это операторы для управления разрешениями, с помощью них мы можем разрешать или запрещать выполнение определенных операций над объектами базы данных.

К операторам DCL относятся:

– GRANT

– REVOKE

– DENY

Для начала определимся с исходными данными, которые будут использовать в примерах.

Представим, что есть база данных TestDB, при этом появился новый пользователь, и для него создали учетную запись, т. е. имя входа и пользователя на SQL сервере. Данная учетная запись не имеет никаких прав в базе данных, она имеет только предопределенную серверную роль public.

В базе данных у есть таблица Goods, она содержит данные о товарах, и процедура TestProcedure. С этими объектами и будем работать.

Все объекты в базе данных Вы можно создать с помощью следующей инструкции.

```
USE TestDB;
– Создание имени входа
CREATE LOGIN TestLogin
  WITH PASSWORD='Pa$$w0rd',
  DEFAULT_DATABASE=TestDB;
```

– Создание пользователя базы данных и сопоставление с именем входа

```
CREATE USER TestUser FOR LOGIN TestLogin;
– Создание таблицы Goods
CREATE TABLE Goods (
  ProductId       INT  IDENTITY(1,1)  NOT  NULL  PRIMARY
KEY,
  Category       INT NOT NULL,
  ProductName    VARCHAR(100) NOT NULL,
  Price          MONEY NULL,
);
– Добавление строк в таблицу Goods
INSERT INTO Goods(Category, ProductName, Price)
  VALUES (1, 'Системный блок', 300),
        (1, 'Монитор', 200),
        (2, 'Смартфон', 100);
– Создание процедуры
CREATE PROCEDURE TestProcedure (@ProductId INT)
AS
BEGIN
  SELECT * FROM Goods
  WHERE ProductId = @ProductId;
END
SELECT * FROM Goods;
```

Оператор GRANT

GRANT – предоставляет пользователю или группе разрешения на определенные операции с объектом.

В качестве объекта может выступать: таблица, представление, функция, хранимая процедура и т. д.

Операции над объектами могут быть разными, например, у таблицы это извлечение данных (SELECT), добавление данных (INSERT), изменение данных (UPDATE), удаление данных (DELETE), а также изменение самой таблицы (ALTER).

На текущий момент у тестового пользователя никаких прав нет, и если подключиться этим пользователем, то при попытке даже простого обращения к таблице получите следующую ошибку.

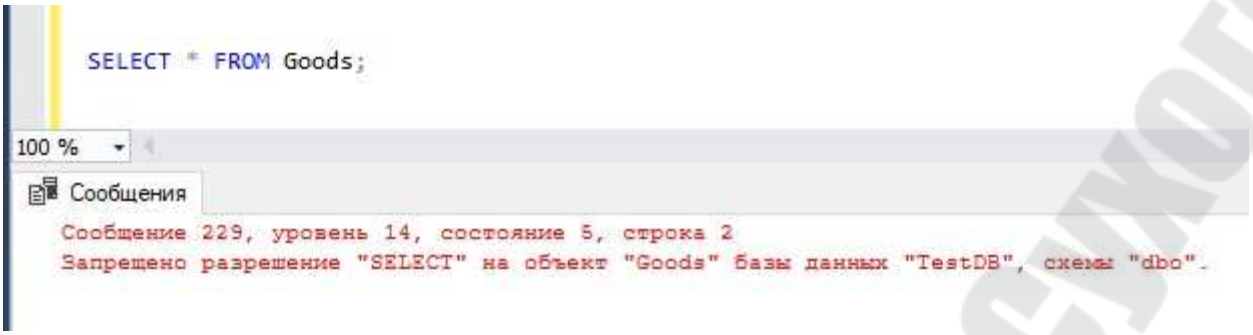

Рис.1.6. На текущий момент у тестового пользователя никаких прав нет

Давайте для примера дадим нашем новому пользователю возможность извлекать, изменять, добавлять и удалять данные из нашей тестовой таблицы.

Для этого можно выполнить вот такую инструкцию.

GRANT SELECT, INSERT, UPDATE, DELETE ON Goods TO TestUser;

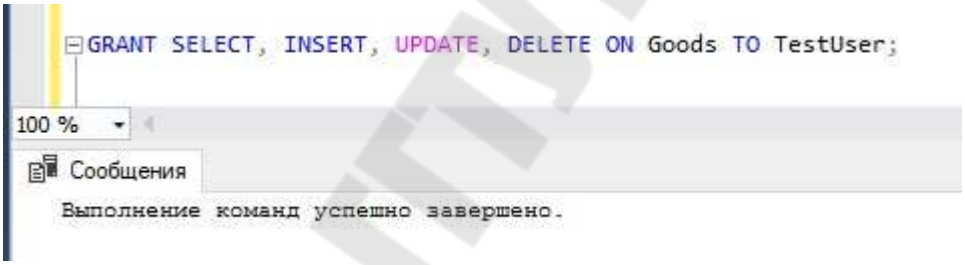

Рис. 1.7. Задаем новому пользователю возможность извлекать, изменять, добавлять и удалять данные из нашей тестовой таблицы

Где после команды GRANT перечислены операции, которые хотим разрешить выполнять пользователю. Затем пишем ключевое слово ON и указываем объект, в данном случае таблицу, на который представляем эти права. Далее пишем ключевое слово TO и указываем группу или, как в нашем случае, конкретного пользователя, которому представляется права.

И если сейчас запустить тот же самый запрос SELECT от имени тестового пользователя, то он выполнится вполне успешно.

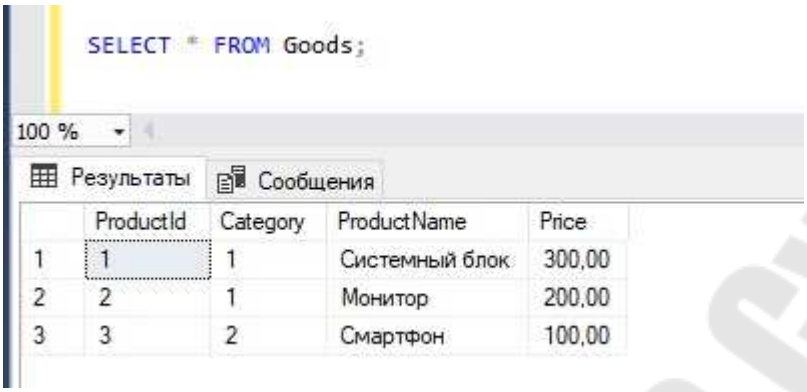

Рис. 1.8. Выполнение запроса SELECT от имени тестового пользователя

Разрешения для работы с таблицей уже есть, однако в тестовой базе данных есть еще и процедура, которую, допустим, пользователь должен запускать.

Однако если он сейчас попытается это сделать, у него выйдет ошибка.

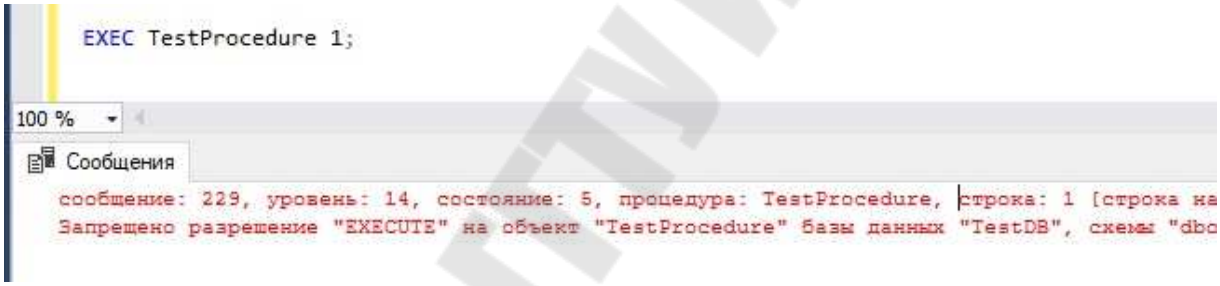

Рис. 1.9. Ошибка при попытке пользователя вызвать тестовую процедуру

Поэтому давайте дадим ему разрешение на запуск этой процедуры. Это делается следующим образом.

GRANT EXECUTE ON TestProcedure TO TestUser;

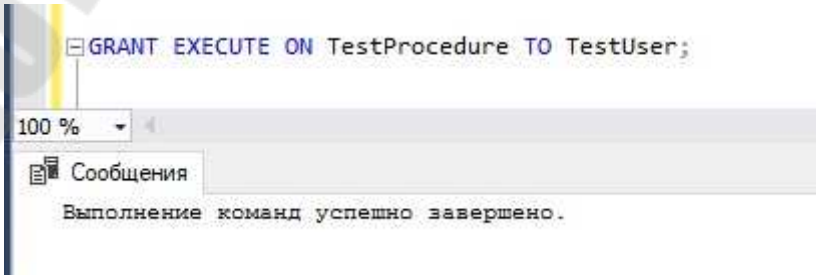

Рис. 1.10. Задание разрешения на запуск тестовой процедуры

Разрешение на запуск процедуры называется EXECUTE, и таким образом, используя тот же самый принцип, что использовали ранее, пишем инструкцию, только в качестве разрешения указываем EXECUTE, а в качестве объекта – название хранимой процедуры.

Оператор REVOKE

REVOKE – отзывает выданные разрешения.

Иным словами, с помощью этого оператора можно отменить выданное или запрещенное ранее разрешение.

Допустим, погорячились с тем, что выдали пользователю разрешение на удаление данных из таблицы, а также с тем, что дали ему возможность запускать хранимую процедуру. И возникала необходимость отменить именно эти разрешения.

Это можно сделать с помощью следующих инструкций.

```
– Отменяем разрешение на удаление данных
REVOKE DELETE ON Goods TO TestUser;
– Отменяем разрешение на запуск процедуры
REVOKE EXECUTE ON TestProcedure TO TestUser;
```

После оператора REVOKE пишем разрешения, которые хотим отменить, с помощью ключевого слова ON указываем объект, а с помощью ключевого слова TO указываем субъект, т. е. пользователя, у которого мы хотим отобрать разрешения.

Оператор DENY

DENY– задает запрет, имеющий приоритет над разрешением.

С помощью DENY явно запрещается выполнение определенных действий даже в том случае, если пользователь состоит в группе или роли, которая имеет разрешение на выполнение этих действий. Таким образом, предотвращается наследование разрешения участником через его членство в группе или роли.

DENY имеет приоритет над всеми разрешениями, но не применяется к владельцам объектов или членам с предопределенной ролью сервера sysadmin, так как им не может быть отказано в разрешениях.

Теперь давайте представим, что нашему пользователю присвоена определенная роль, как обычно и делается, и эта роль имеет право на запуск нашей тестовой процедуры. Однако, категорически нельзя ему разрешать запуск этой процедуры, при этом запретить всей роли

запуск процедуры не можем, так как все ее участники должны ее запускать (но кроме нашего тестового пользователя).

Решить данную проблему поможет оператор DENY, с помощью которого явно можно запретить выполнение хранимой процедуры только нашему тестовому пользователю. Например, следующим образом.

```
DENY EXECUTE ON TestProcedure TO TestUser;
```

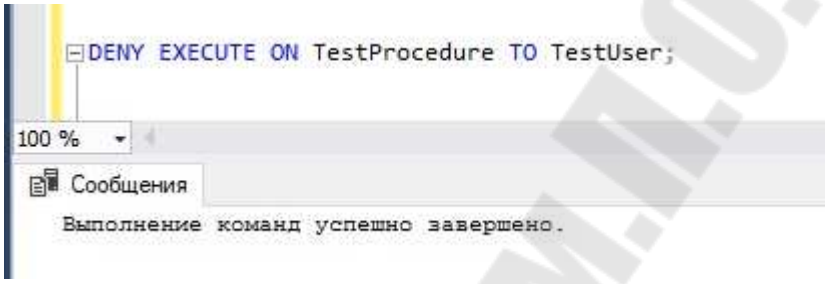

Рис. 1.11. Запрещение выполнение хранимой процедуры только тестовому пользователю

В этом случае делается практически все то же самое, только в начале пишем оператор DENY.

1.6 Введение в OLAP. OLAP против OLTP

Понятие OLAP. Требования Кодда к OLAP-системам. Тест FASMI

Традиционным подходом к организации баз данных и соответствующих обслуживающих приложений является OLTP-подход.

OLTP, или Online Transaction Processing, – это обработка транзакций в реальном времени. Структура такой базы данных сильно нормализована и оптимизирована для выполнения коротких идущих большим потоком транзакций, при этом клиенту требуется от системы минимальное время отклика. Обрабатываемый и сохраняемый OLTP-системой в течение дня объем данных может достигать нескольких гигабайт. Примерами применения OLTP-подхода могут служить системы учета биржевых, банковских операций, системы бухгалтерского и складского учета и т.д.

Благодаря нормализации в таких системах значительно снижается избыточность данных и вычислительные потребности на опера-

ции обновления, что делает OLTP-системы идеальными для хранения данных. Однако сложность структуры таблиц и большие объемы накопленных данных приводят к снижению скорости выполнения сложных запросов на извлечение данных (например, посчитать прибыль организации по кварталам за последние пять лет), снижению производительности системы в целом. В результате эти системы оказываются непригодными для решения задач, диктуемых бизнес-аналитиками.

Поиск решения данной проблемы привел к формированию совершенно нового подхода, получившего название OLAP (Online Analytical Processing) – технология оперативной аналитической обработки данных, использующая методы и средства для сбора, хранения и анализа многомерных данных в целях поддержки процессов принятия решений. Цель таких систем – проверка гипотез пользователя-аналитика.

Основоположником термина «OLAP» является Эдгар Кодд, известный как классик теории реляционных баз данных. В 1993 году он опубликовали статью «Обеспечение OLAP (оперативной аналитической обработки) для пользователей-аналитиков», в которой были изложены 12 законов, заложившие основу концепции аналитической обработки данных в реальном времени. Позднее, в 1995 году эти правила были дополнены еще шестью.

1. Многомерное концептуальное представление данных. Эта особенность – основа технологии OLAP. Вместо привычной модели данных реляционных источников, основанной на плоской системе координат, пользователь получает в свое распоряжение интуитивно понятную многомерную модель, в которой данные организуются в виде многомерных кубов (гиперкубов). Осями многомерной системы координат служат основные атрибуты анализируемого бизнес-процесса (товар, регион, тип покупателя, время и т.д.). На пересечениях осей (измерений) многомерной системы координат находятся данные, количественно характеризующие процесс – меры (объемы, остатки на складе, издержки и т. п.).

2. Интуитивное манипулирование данными.

3. Доступность: OLAP как посредник между гетерогенными источниками данных и представлением для конечного пользователя.

4. Пакетное извлечение против интерпретации. Требуется, чтобы продукт в равной степени эффективно обеспечивал доступ и к собственному хранилищу данных, и к внешним данным.

5. Модели анализа OLAP. Требуется, чтобы OLAP-системы поддерживали формирование настраиваемых отчетов, формирование разрезов и группировок данных, проверку гипотез (ответы на вопрос "что, если…?") и модели поиска целей.

6. Архитектура "клиент-сервер". Требуется также, чтобы серверный компонент был бы достаточно интеллектуальным для того, чтобы различные клиенты могли подключаться с минимумом усилий и программирования.

7. Прозрачность. Это требование означает, что пользователь получает все необходимые данные из OLAP-машины, не подозревая, откуда они берутся.

8. Многопользовательская поддержка. Инструменты OLAP должны обеспечивать одновременный доступ (чтение и запись), интеграцию и конфиденциальность.

9. Обработка ненормализованных данных. Данное требование указывает на необходимость интеграции между OLAP-машиной и ненормализованными источниками данных. Модификации данных, выполненные в среде OLAP, не должны приводить к изменениям данных, хранимых в исходных внешних системах.

10. Сохранение результатов OLAP: хранение их отдельно от исходных данных.

11. Исключение отсутствующих значений. Отсутствующие значения должны отличаться от нулевых значений.

12. Обработка отсутствующих значений. Все отсутствующие значения будут игнорироваться OLAP-анализатором без учета их источника.

13. Гибкость формирования отчетов. Измерения должны быть размещены в отчете так, как это нужно пользователю.

14. Стандартная производительность отчетов. Требуется, чтобы производительность формирования отчетов существенно не падала с ростом количества измерений и размеров базы данных.

15. Автоматическая настройка физического уровня. Требуется, чтобы OLAP-системы автоматически настраивали свою физическую схему в зависимости от типа модели, объемов данных и разреженности базы данных.

16. Универсальность измерений. Все измерения должны быть равноправны, каждое измерение должно быть эквивалентно и в структуре, и в операционных возможностях.

17. Неограниченное число измерений и уровней агрегации. Кодд предлагает, что в случае принятия некоторого максимума, он должен обеспечивать хотя бы 15 измерений, а предпочтительнее – 20.

18. Неограниченные операции между размерностями. Все виды операций должны быть дозволены для любых измерений.

Альтернативным набором критериев определения OLAP является широко известный сформулированный Найджелом Пендсом и Ричардом Критом в 1995 г. тест FASMI, или Fast Analysis of Shared Multidimensional Information – Быстрый Анализ Разделяемой Многомерной Информации:

FAST (Быстрый) – означает, что система должна обеспечивать выдачу большинства ответов пользователям в сжатые сроки; при этом самые простые запросы обрабатываются в течение одной секунды и лишь немногие – более 20;

ANALYSIS (Анализ) – означает, что система может справляться с любым логическим и статистическим анализом, характерным для данного приложения, и обеспечивает его сохранение в виде, доступном для конечного пользователя;

SHARED (Разделяемый) – означает, что система осуществляет все требования защиты конфиденциальности (возможно до уровня ячейки) и, если множественный доступ для записи необходим, обеспечивает блокировку модификаций на соответствующем уровне;

MULTIDIMENSIONAL (Многомерный) – означает, что система должна обеспечить многомерное концептуальное представление данных, включая полную поддержку для иерархий и множественных иерархий; многомерность – ключевой критерий;

INFORMATION (Информация) – требуемая информация должна быть получена там, где она необходима.

Технологии OLAP и OLTP. Особенности, различия

Сопоставим основные характеристики технологий OLAP и OLTP в форме таблицы, чтобы окончательно развести эти два понятия (таблица 1.1).

Таблица 1.1. Сравнительная характеристика технологий OLAP и OLTP

Признаки сравнения	OLAP	OLTP
Объем хранимой информации	Большой объем информации	Большой объем информации
Хранение данных	Синхронизированная информация из различных баз данных с использованием общих классификаторов	Зачастую различные базы данных для отдельных подразделений
Отношение к нормализации	Ненормализованная схема, существуют дубликаты данных	Нормализованная схема, дубликаты данных отсутствуют
Частота изменения данных	Производится редко через пакетную загрузку	Интенсивное изменение данных
Специфика режима работы с данными	Система выполняет сложные ранее не регламентированные запросы над большими объемами данных с широким применением группировок; анализ временных зависимостей	Система работает в транзакционном режиме; транзакции малы по объему обрабатываемой информации; наборы процедур, запросов определены заранее
Пользователи	Малое количество пользователей (менеджеры, аналитики)	Большое количество пользователей-операторов)

Как отмечалось ранее OLTP-системы, приспособленные для хранения данных, оказались непригодными для задач аналитиков. OLAP-системы же оптимизированы для выполнения операций чтения над большими объемами данных. Высокая скорость выполнения сложных аналитических запросов OLAP-системами связана с особенностями построения используемых ими многомерных структур (многомерные базы данных, или OLAP-кубы):

OLAP-системы строятся на базе денормализованных источников – хранилищ данных; в итоге в базе данных OLAP могут содержаться избыточные данные, но в то же время положительным моментом в

упрощении структуры связей таблиц является повышение скорости выполнения запросов; многомерный куб содержит в себе не только сами данные, но и их агрегаты (обобщенные показатели) по различным измерениям; т. е. в базе данных OLAP хранятся заранее посчитанные системой показатели, которые потенциально могут потребоваться бизнес-аналитику.

Многомерность в OLAP-системах можно быть представить на трех уровнях:

– многомерное представление данных – средства на стороне клиента, обеспечивающие многомерную визуализацию и манипулирование данными; слой многомерного представления абстрагирован от физической структуры данных и воспринимает данные как многомерные;

– многомерная обработка – средство (язык) формирования многомерных запросов (традиционный реляционный язык SQL здесь оказывается непригодным) и процессор, умеющий обработать и выполнить такой запрос;

– многомерное хранение – средства физической организации данных, обеспечивающие эффективное выполнение многомерных запросов.

Первые два уровня в обязательном порядке присутствуют во всех OLAP-средствах. Третий уровень, хотя и является широко распространенным, не обязателен, так как данные для многомерного представления могут извлекаться и из обычных реляционных структур; процессор многомерных запросов в этом случае транслирует многомерные запросы в SQL-запросы, которые выполняются реляционной СУБД.

Виды OLAP-серверов

В соответствии с требованием прозрачности OLAP-систем способ реализации многомерной модели скрыт от пользователя. Однако способ реализации важен, поскольку от него зависят производительность решения и требуемые ресурсы. Существует три основных способа реализации многомерной модели: MOLAP, ROLAP, HOLAP

MOLAP (Multidimensional OLAP, или многомерный OLAP) – исходные и агрегатные данные хранятся в многомерной базе данных. Хранение данных в многомерных структурах позволяет манипулировать данными как многомерным упорядоченным массивом, благодаря

чему скорость вычисления агрегатных значений одинакова для любого из измерений. Физически данные хранятся в "плоских" файлах, при этом куб представляется в виде одной плоской таблицы, в которую построчно вписываются все комбинации элементов всех измерений с соответствующими им значениями мер.

В силу своих особенностей использование MOLAP является эффективным при следующих условиях:

– объем исходных данных для анализа не слишком велик (не более нескольких гигабайт), т. е. уровень агрегации данных достаточно высок;

– набор информационных измерений стабилен (MOLAP чувствителен к изменению многомерной модели);

– наименьшее время отклика системы на нерегламентированные запросы является критичным параметром (MOLAP обеспечивает высокую скорость поиска и выборки);

– требуется использование сложных встроенных функций для выполнения вычислений над ячейками куба, возможность написания пользовательских функций (MOLAP легко справляются с задачами включения в информационную модель разнообразных встроенных функций).

ROLAP (Relational OLAP, или реляционный OLAP) – исходные данные остаются в той же реляционной базе данных, где они изначально и находились. Агрегатные же данные помещают в специально созданные для их хранения служебные таблицы в той же базе данных.

Достоинства ROLAP:

– в большинстве случаев корпоративные хранилища данных реализуются средствами реляционных СУБД, и инструменты ROLAP позволяют производить анализ непосредственно над ними;

– в случае переменной размерности задачи, когда изменения в структуру измерений приходится вносить достаточно часто, ROLAP-системы с динамическим представлением размерности являются оптимальным решением, так как в них такие модификации не требуют физической реорганизации БД;

– реляционные СУБД обеспечивают значительно более высокий уровень защиты данных и хорошие возможности разграничения прав доступа.

Недостатки ROLAP:

– меньшая производительность в сравнении с MOLAP. Для обеспечения производительности, сравнимой с MOLAP, реляционные системы требуют тщательной проработки схемы базы данных и настройки индексов.

HOLAP (Hybrid OLAP, или гибридный OLAP) – исходные данные остаются в той же реляционной базе данных, где они изначально находились, а агрегатные данные хранятся в многомерной базе данных. Серверы HOLAP применяют подход ROLAP для разреженных областей многомерного пространства и подход MOLAP – для плотных областей. Серверы HOLAP разделяют запрос на несколько подзапросов, направляют их к соответствующим фрагментам данных, комбинируют результаты, а затем предоставляют результат пользователю.

1.7 Оконные функции

Еще в Microsoft SQL Server 2005 появился интересный функционал – оконные функции. Это функции, которые позволяют осуществлять вычисления в заданном диапазоне строк внутри предложения Select. Для тех, кто не сталкивался с этими функциями возникает вопрос – «Что значит оконные?». Окно – значит набор строк, в рамках которого происходит вычисление. Оконная функция позволяет разбивать весь набор данных на такие окна.

Конечно, все что могут оконные функции возможно реализовать и без них. Однако оконные функции обладают большим преимуществом перед регулярными агрегатными функциями: нет нужды группировать набор данных для расчетов., что позволяет сохранить все строки набора с их уникальными идентификаторами. При этом результаты работы оконных функций просто добавляются к результирующей выборке как еще одно поле.

Основное преимущество использования оконных функций над регулярными агрегатными функциями заключается в следующем: оконные функции не приводят к группированию строк в одну строку вывода, строки сохраняют свои отдельные идентификаторы, а агрегированное значение добавляется к каждой строке.

Окно определяется с помощью инструкции OVER(). Давайте рассмотрим синтаксис этой инструкции:

Оконная функция (столбец для вычислений) OVER ([PARTITION BY столбец для группировки] [ORDER BY столбец для сортировки] [ROWS или RANGE выражение для ограничения строк в пределах группы])

Оконные функции разделяются на: агрегирующие, ранжирующие, смещения.

Для демонстрации работы оконных функций рассмотрем следующую тестовую таблицу:

```
1       CREATE TABLE ForWindowFunc (ID INT, GroupId INT,
2 Amount INT)
3       GO
4
5       INSERT INTO ForWindowFunc (ID, GroupId, Amount)
6        VALUES(1, 1, 100), (1, 1, 200), (1, 2, 150),
7        (2, 1, 100), (2, 1, 300), (2, 2, 200), (2, 2, 50),
         (3, 1, 150), (3, 2, 200), (3, 2, 10);
```

ID	GroupId	Amount
1	1	100
1	1	200
1	2	150
2	1	100
2	1	300
2	2	200
2	2	50
3	1	150
3	2	200

Как видно, здесь три группы в колонке ID и две подгруппы в колонке GroupId с разным количеством элементов в группе.

Чаще всего используется функция суммирования, поэтому демонстрацию проведем именно на ней. Давайте посмотрим, как работает инструкция OVER:

```
1       SELECT ID,
2       Amount,
3       SUM(Amount) OVER() AS SUM FROM ForWindowFunc
```

ID	Amount	Sum
1	100	1310
1	200	1310
2	100	1310
2	300	1310
2	200	1310
2	50	1310
3	150	1310
3	200	1310
3	10	1310

Была использована инструкцию OVER() без предложений. В таком варианте окном будет весь набор данных и никакая сортировка не применяется. SQL Server может поменять порядок отображения, если нет явно заданной сортировки. Поэтому инструкцию OVER() практически никогда не применяют без предложений. Но, обратим наше внимание на новый столбец SUM. Для каждой строки выводится одно и то же значение 1310. Это сквозная сумма всех значений колонки Amount.

Предложение PARTITION BY

Предложение PARTITION BY определяет столбец, по которому будет производиться группировка, и он является ключевым в разбиении набора строк на окна.

Изменим запрос, написанный ранее, так:

```
1       SELECT ID,  Amount,
2       SUM(Amount) OVER(PARTITION BY ID) AS SUM
3FROM ForWindowFunc
```

ID	Amount	Sum
1	100	300
1	200	300
2	100	650
2	300	650

2	200	650
2	50	650
3	150	360
3	200	360
3	10	360

Предложение PARTITION BY сгруппировало строки по полю ID. Теперь для каждой группы рассчитывается своя сумма значений Amount. То есть можно создавать окна по нескольким полям. Тогда в PARTITION BY нужно писать поля для группировки через запятую (например, PARTITION BY ID, Amount).

Предложение ORDER BY

Вместе с PARTITION BY может применяться предложение ORDER BY, которое определяет порядок сортировки внутри окна. Порядок сортировки очень важен, ведь оконная функция будет обрабатывать данные согласно этому порядку. Если не использовать предложение PARTITION BY, а только ORDER BY, то окном будет весь набор данных.

```
1    SELECT ID,
2    GroupId,
3    Amount,
4    SUM(Amount) OVER(PARTITION BY id ORDER BY
     Amount) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	1	100	100
1	2	150	250
1	1	200	450
2	2	50	50
2	1	100	150
2	2	200	350
2	1	300	650
3	2	10	10
3	1	150	160
3	2	200	360

К предложению PARTITION BY добавилось ORDER BY по полю Amount. Таким образом было указано, что необходимо найти сумму не всех значений Amount в окне, а для каждого значения Amount сумму со всеми предыдущими. Такое суммирование часто называют нарастающий итог или накопительный итог.

Также в выборке появилось поле GpoupId. Это поле позволяет показать, как изменится нарастающий итог, в зависимости от сортировки. Изменим запрос:

```
1       SELECT ID,
2        GroupId,
3        Amount,
4        SUM(Amount) OVER(Partition BY id ORDER BY
  GroupId, Amount) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	1	100	100
1	1	200	300
1	2	150	450
2	1	100	100
2	1	300	400
2	2	50	450
2	2	200	650
3	1	150	150
3	2	10	160
3	2	200	360
3	2	200	360

И здесь получается совсем другое поведение. И хоть в итоге для последнего значения в окне значения сходятся с предыдущим примером, но сумма для всех остальных отличается. Поэтому важно четко понимать, что необходимо получить в итоге.

Предложение ROWS/RANG

Еще два предложения ROWS и RANGE применяются в инструкции OVER. Этот функционал появился в MS SQL Server 2012.

Предложение ROWS ограничивает строки в окне, указывая фиксированное количество строк, предшествующих или следующих за

текущей. Оба предложения ROWS и RANGE используются вместе с ORDER BY.

Предложение ROWS может быть задано с помощью методов:

– CURRENT ROW – отображение текущей строки;

– UNBOUNDED FOLLOWING – все записи после текущей;

– UNBOUNDED PRECEDING – все предыдущие записи;

– <целое число> PRECEDING – заданное число предыдущих строк;

– <целое число> FOLLOWING – заданное число последующих записей.

Можно комбинировать эти функции для достижения желаемого результата, например:

ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING – в окно попадут текущая и одна следующая запись;

```
1       SELECT ID,
2        GroupId,
3        Amount,
4        SUM(Amount) OVER(Partition BY id ORDER BY
  GroupId, Amount ROWS BETWEEN CURRENT ROW AND 1
  FOLLOWING ) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	1	**100**	**300**
1	1	**200**	350
1	2	150	150
2	1	100	400
2	1	300	350
2	2	50	250
2	2	200	200
3	1	150	160
3	2	10	210
3	2	200	200

Здесь, сумма рассчитывается по текущей и следующей ячейке в окне. А последняя в окне строка имеет то же значение, что и Amount. Посмотрим на первое окно, выделенное жирным. Сумма 300 рассчитана сложением 100 и 200. Для следующего значения ситуация анало-

гичная. А последняя в окне сумма имеет значение 150, потому что текущий Amount больше не с чем складывать.

ROWS BETWEEN 1 PRECEDING AND CURRENT ROW – одна предыдущая и текущая запись

```
1   SELECT ID,
2    GroupId,
3    Amount,
4   SUM(Amount) OVER(Partition BY id ORDER BY
  GroupId, Amount ROWS BETWEEN 1 PRECEDING AND
  CURRENT ROW) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	1	**100**	100
1	1	**200**	**300**
1	2	150	350
2	1	100	100
2	1	300	400
2	2	50	350
2	2	200	250
3	1	150	150
3	2	10	160
3	2	200	210

В этом запросе сумма получается путем сложения текущего значения Amount и предыдущего. Первая строка имеет значение 100, т.к. предыдущего значения Amount не существует.

Предложение RANGE предназначено также для ограничения набора строк. В отличие от ROWS, оно работает не с физическими строками, а с диапазоном строк в предложении ORDER BY. Это означает, что одинаковые по рангу строки в контексте предложения ORDER BY будут считаться как одна текущая строка для функции CURRENT ROW. А в предложении ROWS текущая строка – это одна, текущая строка набора данных.

Предложение RANGE может использоваться только с опциями CURRENT ROW, UNBOUNDED PRECEDING и UNBOUNDED FOLLOWING.

Предложение RANGE может использовать опции:

– CURRENT ROW – отображение текущей строки;

– UNBOUNDED FOLLOWING – все записи после текущей;

– UNBOUNDED PRECEDING – все предыдущие записи.

И не может:

– <целое число> PRECEDING – заданное число предыдущих строк;

– <целое число> FOLLOWING – заданное число последующих записей.

Примеры:

RANGE CURRENT ROW;

```
1       SELECT ID,
2        GroupId,
3        Amount,
4        SUM(Amount) OVER(Partition BY id ORDER BY GroupId
  RANGE CURRENT ROW) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	**1**	**100**	**300**
1	**1**	**200**	**300**
1	2	150	150
2	1	100	400
2	1	300	400
2	2	200	250
2	2	50	250
3	1	150	150
3	2	200	210
3	2	10	210

Предложение Range настроено на текущую строку. Но, для Range текущая строка, это все строки, соответствующие одному значению сортировки. Сортировка в данном случае по полю GroupId. Первые две строки первого окна имеют значение GroupId равное 1 – следовательно оба эти значения удовлетворяют ограничению RANGE CURRENT ROW. Поэтому Sum для каждой из этих строк равна общей сумме Amount по ним - 300.

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW;

```
1       SELECT ID,
```

```
2       GroupId,
3       Amount,
4       SUM(Amount) OVER(Partition BY id ORDER BY GroupId
  RANGE BETWEEN UNBOUNDED PRECEDING AND
  CURRENT ROW) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	1	**100**	300
1	1	**200**	300
1	2	**150**	**450**
2	1	100	400
2	1	300	400
2	2	200	650
2	2	50	650
3	1	150	150
3	2	200	360
3	2	10	360

В этом случае ограничение по всем предыдущим строкам и текущей. Для первой и второй строки это правило работает как предыдущее (вспоминаем CURRENT ROW), а для третьей как сумма Amount предыдущих строк с текущей.

RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

```
1       SELECT ID,
2       GroupId,
3       Amount,
4       SUM(Amount) OVER(Partition BY id ORDER BY GroupId
  RANGE BETWEEN CURRENT ROW AND UNBOUNDED
  FOLLOWING) AS SUM FROM ForWindowFunc
```

ID	GroupId	Amount	Sum
1	2	**150**	150
1	1	**100**	**450**
1	1	**200**	450
2	2	200	250
2	2	50	250

2	1	100	650
2	1	300	650
3	2	200	210
3	2	10	210
3	1	150	360

Это ограничение позволяет получить сумму из текущей строки и всех предыдущих в рамках одного окна. Так как вторая и третья строка находятся в одной GroupId, то эти значения и есть Current Row. Поэтому они просуммированы сразу.

Лучше всего понять суть оконных функций можно на примере. Допустим у вас есть данные о платежах абонентов. Платеж поступает на договор. Но у этого договора есть дочерние договора, на которых отрицательный баланс. И мы хотим распределить поступившие средства на погашение долга дочерних договоров.

Таким образом, нам нужно определить сколько же денег мы спишем со счета основного договора и сколько переведем на дочерний. Давайте посмотрим на таблицу:

ContractId	**ChildContractId**	**PayId**	**CustAccount**	**PayAmount**
1000000000	1000000002	1000000752	-200,00	800,00
1000000000	1000000003	1000000753	-1000,00	800,00

где, ContractId – идентификатор основного договора,

ChildContractId – идентификатор дочернего договора,

PayId – идентификатор платежа,

CustAccount – баланс дочернего договора,

PayAmount – платеж.

Из таблицы видно, что для каждого дочернего договора сумма платежа 800. Это из-за того, что платеж на родительском договоре.

Так что наша задача рассчитать суммы переносов денег с родительского на дочерние договора.

Для этого суммируем **CustAccount** и **PayAmount.** Однако, простая сумма баланса и платежа не нас не устроит. Ведь на погашение долга на втором дочернем договоре необходимо учитывать остаток от суммы баланса первого договора и платежа.

Как же действовать в этой ситуации? Можно выбрать:

```
1       SELECT
```

```
2       ContractId,
3       ChildContractId,
4       PayId,
5       CustAccount,
6       PayAmount,
7       PayAmount + (SELECT SUM(CustAccount) FROM
8   dbo.Pays p2 WHERE p1.PayId = p2.PayId AND
    p2.ChildContractId <= p1.ChildContractId) AS [SUM]
        FROM dbo.Pays p1
```

Этот запрос решает поставленную задачу, но подзапрос портит всю картину – увеличивает время выполнения запроса. Применим оконную функцию сложения:

```
1       SELECT
2       ContractId,
3       ChildContractId,
4       PayId,
5       CustAccount,
6       PayAmount,
7       PayAmount + SUM(CustAccount) OVER (ORDER BY
8   ChildContractId) AS [SUM]
        FROM dbo.Pays p1
```

Этот вариант работает быстрее и выглядит лаконичнее. В данном случае получаем сумму по полю CustAccount в окне, которое формируется по полю ChildContractId.

Результатом этих запросов будет таблица:

ContractId	ChildContractId	PayId	CustAccount	PayAmount	Sum
1000000000	1000000002	1000000752	-200,00	800,00	600
1000000000	1000000003	1000000753	-1000,00	800,00	-400

Исходя из полученных данных в колонке Sum определяем сумму, которую нужно перенести с родительского договора на дочерний. Для договора 1000000002 погашаем долг полностью, так что сумма платежа 200р. Для договора 1000000003 долг погашен частично – сумма платежа равна сумме баланса и остатка от платежа после расчета для первой записи (-1000 + 600 = -400р).

2. DWH и качество данных

2.1 Основы DWH

DWH – что это и в чем отличие от баз данных

Data warehouse – склад всех нужных и важных для принятия решений данных компании.

Может возникнуть вопрос: «Но есть же всякие базы данных внутри фирмы, разве они не DWH? Например, СУБД с клиентами, складскими запасами или покупками. Где разница между обычной базой данных и DWH?»

Разница вот в чем:

– Типы хранимых данных. Обычные СУБД хранят данные строго для определенных подсистем. База данных склада хранит складские запасы и ничего более. База данных кадровиков хранит данные по персоналу, но не товары или сделки. DWH, как правило, хранит информацию разных подразделений – там найдутся данные и по товарам, и по персоналу, и по сделкам.

– Объемы данных. Обычная БД, которая ведется в рамках стандартной деятельности компании, содержит только актуальную информацию, нужную в данный момент для функционирования определенной системы. В DWH пишутся не столько копии актуальных состояний, сколько исторические данные и агрегированные значения. Например, состояние запасов разных категорий товаров на конец смены за последние пять лет. Иногда в DWH пишутся и более крупные пачки данных, если они имеют критическое значение для бизнеса – допустим, полные данные по продажам и сделкам. То есть, по сути, это копия СУБД отдела продаж.

– Место в рабочих процессах. Информация обычно сразу попадает в рабочие базы данных, а уже оттуда некоторые записи переползают в DWH. Склад данных, по сути, отражает состояние других БД и процессов в компании уже после того, как вносятся изменения в рабочих базах.

Короче говоря, DWH – это система данных, отдельная от оперативной системы обработки данных. В корпоративных хранилищах в удобном для анализа виде хранятся архивные данные из разных, иногда очень разнородных источников. Эти данные предварительно обрабатываются и загружаются в хранилище в ходе процессов извлече-

ния, преобразования и загрузки, называемых ETL. Решения ETL и DWH – это (упрощенно) одна система для работы с корпоративной информацией и ее хранения.

Что дают DWH-решения для BI и принятия решений в компании

Понятное дело, что просто так тратить деньги и время на консервирование кучи разных записей, которые и так можно накопать в других базах данных, никто не станет. Ответ заключается в том, что DWH необходима для того, чтобы делать BI – business intelligence.

Что такое BI с DWH? Бизнес-аналитика (BI) – это процесс анализа данных и получения информации, помогающей компаниям принимать решения.

Допустим, в онлайн-магазине упала выручка. Менеджеры зовут на помощь бизнес-аналитика и просят его разобраться. Тот идет в DWH, вынимает оттуда данные по продажам, выручке, количеству пользователей, расходам – и собирает отчет, который в подробностях и с цифрами говорит о причинах падения финансовых показателей. Менеджеры внимательно смотрят на эту информацию и принимают решения по реорганизации ассортимента товаров и маркетинговых политик.

Если бы такого аналитического отчета не было – управленцам пришлось бы искать проблему наугад.

Логичный вопрос: казалось бы, зачем держать для этого всего DWH? Аналитики вполне могут ходить в базы данных разных систем и просто выдергивать оттуда то, что им надо.

Ответ: так, конечно, тоже можно делать. Но – не нужно. И вот почему:

– Доступ к нужным данным. Если компания большая, на получение данных из разных источников нужно собирать разрешения и доступы. У каждого подразделения в такой ситуации, как правило, свои базы данных со своими паролями, которые надо будет запрашивать отдельно. В DWH все нужное уже будет под рукой в готовом виде. Можно просто пойти и дернуть там необходимую статистику.

– Сохранность нужных данных. Данные в DWH не теряются и хранятся в виде, удобном для принятия решений: есть исторические записи, есть агрегированные значения. В операционной базе данных такой информации может и не быть. Например, админы уж точно не

будут хранить на складском сервере архив запасов за 10 лет – БД склада в таком случае была бы слишком тяжелой. А вот хранить агрегированные запасы со склада в DWH – это нормально.

– Устойчивость работы бизнес-систем. DWH оптимизируется для работы аналитиков, а эти ребята могут запрашивать очень большие объемы информации. Если они будут делать это с помощью DWH – ничего страшного, даже если их запрос будет обрабатываться очень долго. А если запросить слишком много записей с боевой базы данных сервера – он может уйти в отказ до конца выполнения запроса от аналитики и создать проблемы для других систем. DWH исключает риск того, что аналитики что-то повесят или сломают.

Для работы с большими данными используют различные решения, обрабатывающие информацию из DWH. SAS, Mail.ru Cloud Solutions и другие компании предлагают различные варианты коробочных и облачных решений под такие задачи.

DWH и бизнес-аналитики переводят управление компаниями из искусства в науку. Имея под рукой результаты измерений по сотням показателей, можно выдвигать гипотезы и ставить эксперименты. Правильные решения легко подтверждаются объективными цифрами, которые достают аналитики из DWH.

Оптимальные управленческие решения – это не всегда максимизация прибыли. Это еще и выращивание новых производственных мощностей, минимизация негативного влияния на экологию, достойное качество жизни сотрудников, лояльность клиентов и стабильность бизнеса в долгосрочной перспективе. Все эти, казалось бы, сложные и эфемерные показатели можно анализировать с помощью BI и данных из DWH.

Без DWH и аналитиков управление бизнесом превращается в слепую езду по льду – возможно, при определенной сноровке вы попадете куда надо, но шансов улететь в сугроб или в столб все же куда больше.

Обзор гибких методологий проектирования DWH

Многое в жизни проекта зависит от того, насколько хорошо продумана объектная модель и структура базы на старте.

Общепринятым подходом были и остаются различные варианты сочетания схемы “звезда” с третьей нормальной формой. Как правило, по принципу: исходные данные – 3NF, витрины – звезда. Этот

подход, проверенный временем и подкрепленный большим количеством исследований – первое (а иногда и единственное), что приходит в голову опытному DWH-шнику при мысли о том, как должно выглядеть аналитическое хранилище.

С другой стороны – бизнесу в целом и требованиям заказчика в частности свойственно быстро меняться, а данным – расти как “вглубь”, так и “вширь”. И вот тут проявляется основной недостаток звезды – ограниченная гибкость.

Для начала давайте определимся, какими свойствами должна обладать система, чтобы ее можно было назвать “гибкой”. Отдельно стоит оговориться, что описываемые свойства должны относиться именно к системе, а не к процессу ее разработки.

И так, какими же возможностями должно обладать гибкое хранилище? Тут можно выделить три пункта:

1. Ранняя поставка и быстрая доработка – это значит, что в идеале первый бизнес-результат (например, первые работающие отчеты) должен быть получен как можно раньше, то есть еще до того, как полностью спроектирована и внедрена система целиком. При этом каждая следующая доработка тоже должна занимать как можно меньше времени.

2. Итеративная доработка – это значит, что каждая следующая доработка в идеале не должна затрагивать уже работающий функционал. Именно этот момент часто становится самым большим кошмаром на крупных проектах – рано или поздно отдельные объекты начинают обрастать таким количеством связей, что становится проще полностью повторить логику в копии рядом, чем добавить поле в существующую таблицу.

3. Постоянная адаптация к меняющимся требованиям бизнеса – общая объектная структура должна быть спроектирована не просто с учетом возможного расширения а с расчетом на то, что направление этого очередного расширения не могло бы вам даже присниться на этапе проектирования.

И да, соответствие всем этим требованиям в одной системе возможно (разумеется, в определенных случаях и с некоторыми оговорками).

Проблемы "классического" подхода и их решения в гибких методологиях

1. Жесткая кардинальность связей.

В основу такой модели закладывается четкое разделение данных на измерения (Dimension) и факты (Fact). При этом связи между объектами закладываются в виде связей между таблицами по внешнему ключу. Это выглядит вполне естественно, но сразу же приводит к первому ограничению гибкости – жесткому определению кардинальности связей.

Это значит, что на этапе проектирования таблиц необходимо точно определить для каждой пары связанных объектов могут ли они относиться как многие-ко-многим, или только 1-ко-многим, и "в какую сторону". От этого напрямую зависит в какой из таблиц будет первичный ключ а в какой – внешний. Изменение этого отношения при получении новых требований с большой вероятностью приведет к переработке базы.

Избежать такой ситуации оказалось довольно просто: для этого достаточно все связи изначально хранить в отдельных таблицах и обрабатывать как многие-ко-многим.

Такой подход был предложен Дэном Линстедтом (Dan Linstedt) как часть парадигмы Data Vault и полностью поддержан Ларсом Реннбэком (Lars Rönnbäck) в Якорной Модели (Anchor Model).

В итоге получаем первую отличительную особенность гибких методологий:

Связи между объектами не хранятся в атрибутах родительских сущностей, а представляют собой отдельный тип объектов.

2. Дублирование данных.

Вторая проблема, решаемая гибкими архитектурами, менее очевидна и свойственна, в первую очередь, измерениям типа SCD2 (медленно меняющиеся измерения второго типа), хотя и не только им.

В классическом хранилище измерение обычно представляет собой таблицу, которая содержит суррогатный ключ (в качестве PK) а также набор бизнес-ключей и атрибутов в отдельных колонках.

Если измерение поддерживает версионность, к стандартному набору полей добавляются границы времени действия версии, а на одну строку в источнике появляется несколько версий в хранилище (по одной на каждое изменение версионных атрибутов).

Если измерение содержит хотя бы один часто изменяющийся версионный атрибут, количество версий такого измерения будет внушительным (даже если остальные атрибуты не версионные, или никогда не изменяются), а если таких атрибутов несколько – количество версий может расти в геометрической прогрессии от их количества. Такое измерение может занимать существенный объем дискового пространства, хотя большая часть хранящихся в нем данных – просто дубли значений неизменных атрибутов из других строк.

3. Нелинейная сложность доработки.

При этом каждая новая витрина, строящаяся на основании другой, увеличивает количество мест, в которых данные могут “разойтись” при внесении изменений в ETL. Это, в свою очередь, приводит к возрастанию сложности (и длительности) каждой следующей доработки.

Если вышеописанное касается систем с редко дорабатываемыми ETL-процессами, жить в такой парадигме можно – достаточно просто следить за тем, чтобы новые доработки корректно вносились во все связанные объекты. Если же доработки происходят часто, вероятность случайно “упустить” несколько связей существенно возрастает.

Если вдобавок учесть, что “версионный” ETL существенно сложнее, чем “не версионный”, избежать ошибок при частой доработке всего этого хозяйства становится достаточно сложно.

Хранение объектов и атрибутов в Data Vault и Anchor Model

Подход, предлагаемый, авторами гибких архитектур, можно сформулировать так:

Необходимо отделить то, что изменяется, от того, что остается неизменным. То есть хранить ключи отдельно от атрибутов.

При этом не стоит путать не версионный атрибут с неизменным: первый не хранит историю своего изменения, но может меняться (например, при исправлении ошибки ввода или получении новых данных) второй – не меняется никогда.

Точки зрения на то, что именно можно считать неизменным в Data Vault и Якорной модели расходятся.

С точки зрения архитектуры Data Vault, неизменным можно считать весь набор ключей – натуральные (ИНН организации, код товара в системе-источнике и т. п.) и суррогатные. При этом остальные атрибуты можно разделить по группам по источнику и/или частоте

изменений и для каждой группы вести отдельную таблицу с независимым набором версий.

В парадигме же Anchor Model неизменным считается только суррогатный ключ сущности. Все остальное (включая натуральные ключи) – просто частный случай его атрибутов. При этом все атрибуты по умолчанию независимы друг от друга, поэтому для каждого атрибута должна быть создана отдельная таблица.

В Data Vault таблицы, содержащие ключи сущностей, называются Хабами (Hub). Хабы всегда содержат фиксированный набор полей:

1. Натуральные ключи сущности.
2. Суррогатный ключ.
3. Ссылку на источник.
4. Время добавления записи.

Записи в Хабах никогда не изменяются и не имеют версий. Внешне хабы очень похожи на таблицы типа ID-map, применяемые в некоторых системах для генерации суррогатов, однако в качестве суррогатов в Data Vault рекомендуется применять не целочисленный сиквенс, а хэш от набора бизнес-ключей. Такой подход упрощает загрузку связей и атрибутов из источников, но может вызвать другие проблемы (связанные, например, с коллизиями, регистром и непечатными символами в строковых ключах и т.п.), поэтому не является общепринятым.

Все остальные атрибуты сущностей хранятся в специальных таблицах, называемых Сателлитами (Satellit). Один хаб может иметь несколько сателлитов, хранящих разные наборы атрибутов.

В Якорной модели (Anchor Model) таблицы, хранящие ключи, называются Якорями (Anchor). И хранят они:

1. Только суррогатные ключи.
2. Ссылку на источник.
3. Время добавления записи.

Натуральные ключи с точки зрения Якорной Модели считаются обычными атрибутами. Такой вариант может показаться более сложным для понимания, но он дает намного больше простора для идентификации объекта.

Например, если данные об одной и той же сущности могут поступать из разных систем, в каждой из которых используется свой натуральный ключ. В Data Vault это может приводить к достаточно гро-

моздким конструкциям из нескольких хабов (по одному на источник + объединяющая мастер-версия), в Якорной модели же натуральный ключ каждого источника попадает в свой атрибут и может использоваться при загрузке независимо от всех остальных.

Еще одно важное различие Data Vault и Якорной модели состоит в наличии атрибутов у связей:

В Data Vault Связи являются таким же полноценными объектами, как и Хабы, и могут иметь собственные атрибуты. В Якорной модели Связи используются только для соединения Якорей и собственных атрибутов иметь не могут.

Как достигается гибкость

Получившаяся конструкция в обоих случаях содержит существенно больше таблиц, чем традиционное измерение. Но может занимать существенно меньше дискового пространства при том же наборе версионных атрибутов, что и традиционное измерение. Никакой магии тут, естественно, нет – все дело в нормализации. Распределяя атрибуты по Сателлитам (в Data Vault) или отдельным таблицам (Anchor Model), уменьшается (или совсем исключается) дублирование значений одних атрибутов при изменении других.

Для Data Vault выигрыш будет зависеть от распределения атрибутов по Сателлитам, а для Якорной модели – практически прямо пропорционален среднему количеству версий на объект измерения.

Однако выигрыш по занимаемому месту – важное, но не главное преимущество отдельного хранения атрибутов. Вместе с отдельным хранением связей, такой подход делает хранилище модульной конструкцией. Это значит, что добавление как отдельных атрибутов, так и целых новых предметных областей в такой модели выглядит как надстройка над существующим набором объектов без их изменения. И это именно то, что делает описанные методологии гибкими.

Также это напоминает переход от штучного производства к массовому – если в традиционном подходе каждая таблица модели уникальна и требует отдельного внимания, то в гибких методологиях – это уже набор типовых “деталей”. С одной стороны, таблиц становится больше, процессы загрузки и выборки данных должны выглядеть сложнее. С другой – они становятся типовыми. А значит, могут быть автоматизированы и управляться метаданными. Вопрос “как будем укладывать?”, ответ на который мог занимать существенную часть

работ по проектированию доработок, теперь просто не стоит (как и вопрос о влиянии изменения модели на работающие процессы).

Это не значит, что аналитики в такой системе совсем не нужны – кто-то все еще должен проработать набор объектов с атрибутами и разобраться откуда и как все это загружать. Но объем работ, а также вероятность и цена ошибки существенно снижаются. Как на этапе анализа, так и при разработке ETL, которая в существенной части может свестись к редактированию метаданных.

2.2 Качество данных для приема (пакетные и потоковые)

Как управлять качеством данных: процессы и инструменты

За оценку качества данных отвечают инженеры Data Quality, которые управляют информационными массивами, проверяют их поведение в текущих и новых условиях, контролируют релевантность, достаточность и актуальность. Как правило, обязанности Data Quality инженера не ограничиваются только рутинными проверками записей в таблицах СУБД, а требуют глубокого понимания бизнес-потребностей, чтобы трансформировать имеющиеся данные в пригодную к практическому использованию информацию. Для этого решаются следующие задачи:

– автоматизированная подготовка тестовых данных;

– загрузка подготовленного датасета в исходный источник, например, озеро данных (Data Lake);

– запуск ETL-процессов для обработки набора данных из исходного источника и отправку в окончательное или промежуточное хранилище с возможностью конфигурации параметров ETL-задачи, например, с помощью Apache Airflow;

– верификация данных после ETL-обработки на предмет их качества и соответствие бизнес-требованиям.

Для организации data chain – цепочки проверочных тестов на каждой стадии обработки данных от источника до пункта финального использования могут использоваться легковесные SQL-запросы. Они помогают оценить отдельные атрибуты качества данных, например, tables metadata, blank lines, NULLs, Errors in syntax. Для регрессионного тестирования, когда используются уже готовые неизменяемые датасеты, код автотестов уже хранит готовые шаблоны проверки данных на соответствие качеству, такие как описания ожидаемых мета-

данных таблиц, строчных выборочных объектов для случайного выбора и т. д. Иногда в ходе тестирования Big Data Quality инженер пишет тестовые ETL-процессы с помощью Apache Spark или Airflow, используя уже готовые операторы, в частности, GCP BigQuery или создавая собственные.

Разумеется, все это инженеры Data Quality выполняют не вручную. Современный рынок ПО предлагает множество специализированных инструментов для проверки качества данных и их улучшения. В частности, Informatica Data Quality, Microsoft Data Quality Services, Oracle Enterprise Data Quality, SAP Data Services, Talend Open Studio for Data Quality и другие коммерческие продукты, а также открытые сервисы.

Как правило, большинство специализированных систем управления качеством данных автоматизируют следующие процессы Data Quality Management:

– профилирование – первоначальная оценка данных, чтобы понять их текущее состояние, в т.ч. распределение значений;

– стандартизация – механизм бизнес-правил, обеспечивающий соответствие данных стандартам;

– геокодирование адресов, которое корректирует данные в соответствии с географическими стандартами;

– сопоставление или связывание – способ сравнения данных для выявления одинаковых по смыслу, но разных по виду представления записей. Сопоставление может использовать нечеткую логику для поиска дубликатов в данных. Например, «Петр» и «Птер» может быть одним и тем же человеком, который проживает по одинаковому адресу.

– мониторинг – отслеживание качества данных с течением времени и отчетность об изменениях с автоматическим исправлением изменений на основе предварительно определенных бизнес-правил;

– пакетная и потоковая обработка – первичная очистка данных в пакетном режиме с последующей интеграцией в корпоративные приложения.

Обеспечение качества данных не сводится лишь к технической задаче устранения дублирующихся или пропущенных значений. Важна также организационная сторона этого процесса, где задействован не только Data Quality инженер.

Пакетная передача данных

Вся информация, передаваемая по сети: файлы, звук, видео и т. д., представляет собой массив цифровых данных. На исходном сервере эти данные (например, HTML-код просматриваемой вами страницы) «разрезаются» на отдельные «порции» заранее оговоренной длины (например, по 256 байт), причем каждая из них снабжается индивидуальным «заголовком». Такая «порция» и называется пакетом.

В «заголовке» содержится информация о месте назначения (адрес, под которым компьютер пользователя, запросившего этот файл, числится в Интернете), об имени файла, к которому принадлежит этот пакет, и о порядковом номере данного пакета (то есть о том, из какого места данного файла он был «вырезан»), а также контрольная сумма - некое число, служащее для проверки правильнос ти передачи.

Пакеты пересылаются по сети Интернет, иногда даже по разным маршрутам, зависящим от загруженности тех или иных линий связи. Маршрут следования каждого пакета определяют специальные компьютеры – IP-маршрутизаторы. Такая технология передачи данных называется динамической маршрутизацией. На пользовательском компьютере для каждого пакета после его получения подсчитывается отдельно друг от друга контрольная сумма и сверяется с тем значением, которое хранится в заголовке. Если два значения контрольной суммы совпадают, то пакет считается принятым без ошибок. В противном случае он повторно запрашивается с сервера (только этот пакет, а не весь файл целиком!). Когда же все пакеты «в сборе», они автоматически объединяются в файл, являющийся точной копией исходного.

Потоковоя передача данных

Потоковые данные – это данные, непрерывно генерируемые тысячами источников данных, которые обычно отправляют записи данных одновременно и небольшими объемами (по несколько килобайтов). В состав потоковых данных входят различные виды данных, например файлы журналов, сформированных клиентами при использовании мобильных или интернет-приложений, покупки в интернет-магазинах, действия игроков в играх, информация из социальных сетей, финансовые торговые площадки и геопространственные сервисы,

а также телеметрические данные, полученные от подключенных устройств или оборудования в ЦОД.

Эти данные должны быть обработаны последовательно и инкрементно либо по каждой из записей, либо с использованием скользящего временного окна, после чего их можно использовать в различных аналитических задачах, включая корреляцию, агрегацию, фильтрацию и шаблонизацию. Информация, полученная в результате подобного анализа, позволяет компаниям разобраться во многих аспектах своей деятельности, например в использовании сервисов (для задач учета/выставления счетов), активности серверов, навигации по веб-сайтам, геолокации устройств, людей или товаров, и в результате быстро реагировать на изменяющиеся условия. К примеру, компании могут отслеживать изменения общественного настроя в отношении своих торговых марок и продуктов за счет постоянного анализа потоков данных из социальных сетей, а в случае необходимости принимать своевременные меры.

Преимущества потоковой передачи данных

Обработка потоковых данных является предпочтительной для большинства сценариев использования, подразумевающих непрерывное формирование новых динамических данных. Обработка потоковых данных применима в большинстве отраслевых сегментов и случаев использования, подразумевающих обработку больших данных. Обычно компании начинают с простых задач, например со сбора данных системных журналов, или с элементарных вычислений, например с обновления минимумов и максимумов. Затем эти задачи трансформируются в более сложную обработку, происходящую в режиме, близком к реальному времени. Изначально приложения могут обрабатывать потоки данных с целью формирования простых отчетов и выполнения простых ответных действий, например активации сигнализации, когда значения ключевых параметров выйдут за указанные границы. В итоге этим приложениям приходится выполнять более сложные формы анализа данных, например применять алгоритмы машинного обучения и достигать более глубокого понимания ситуации на основании данных. С течением времени в этот процесс также добавляются комплексные алгоритмы обработки потоков и событий, например анализ временных окон для определения самых свежих по-

пулярных кинофильмов, что позволяет получать еще более полезную аналитическую информацию.

Примеры потоковой передачи данных

Датчики, используемые в транспортных средствах, промышленном оборудовании и сельскохозяйственной технике, отправляют данные в потоковое приложение. Приложение осуществляет мониторинг производительности, предупреждает возникновение возможных дефектов и автоматически заказывает необходимые запасные части для предотвращения простоя оборудования.

Финансовое учреждение отслеживает изменения на фондовых биржах в режиме реального времени, вычисляет рисковую стоимость и автоматически выполняет ребалансировку портфеля ценных бумаг на основании изменений биржевого курса.

Веб-сайт агентства недвижимости отслеживает набор данных, полученный с мобильных устройств клиентов, и предоставляет рекомендации по объектам недвижимости в режиме реального времени на основании данных геолокации.

Гелиоэнергетическая компания должна предоставлять своим клиентам определенную проходную мощность, в противном случае ей придется платить штрафы. Она развернула приложение для обработки потоковых данных, которое осуществляет мониторинг всех используемых солнечных батарей и планирует необходимое обслуживание в режиме реального времени, что позволяет минимизировать периоды генерации низкой проходной мощности для каждой из батарей и избежать уплаты штрафов.

Мультимедийный издатель осуществляет потоковую передачу миллиардов записей со своих онлайновых ресурсов, выполняет агрегацию и дополнение данных с учетом демографической информации о пользователях и оптимизирует размещение контента на веб-сайте, благодаря чему обеспечивается релевантность контента и повышается качество обслуживания посетителей.

Компания-разработчик интернет-игр выполняет сбор потоковых данных о взаимодействиях игроков с играми и передает эти данные на игровую платформу. Затем выполняется анализ этих данных в режиме реального времени, в результате чего формируются стимулы и динамические эффекты для повышения вовлеченности игроков.

Сравнение пакетной и потоковой обработки

Перед началом работы с потоковой передачей данных стоит сравнить потоковую обработку с пакетной обработкой таблица 2.1 и выявить различия. Пакетная обработка может использоваться для вычислений по любым запросам к данным из разных наборов. Обычно при расчете результатов подобной обработки используются все входящие в пакет данные, благодаря чему обеспечивается глубокий анализ наборов больших данных. В качестве примера платформ, поддерживающих пакетные задания, можно привести системы, использующие MapReduce, например Amazon EMR. В то же время потоковая обработка требует подачи последовательностей данных и инкрементного обновления метрик, отчетов и итоговой статистики в ответ на каждую поступающую запись данных. Этот тип обработки лучше всего подходит для мониторинга в режиме реального времени и функций ответа.

Таблица 2.1. Сравнение пакетной и потоковой обработок

	Пакетная обработка	Потоковая обработка
Охват данных	Запросы ко всем или большей части данных в наборе и их обработка.	Запросы к данным и их обработка в пределах скользящего временного окна или только по самой свежей записи данных.
Размер данных	Большие пакеты данных.	Отдельные записи или микропакеты из нескольких записей.
Производительность	Задержки от нескольких минут до нескольких часов.	Требуется задержка в пределах нескольких секунд или миллисекунд.
Анализ	Комплексная аналитика.	Простые функции ответа, агрегации данных или динамических метрик.

Многие организации выстраивают гибридные модели за счет комбинации двух подходов и поддерживают операции как на уровне реального времени, так и на пакетном уровне. Сначала данные обрабатываются с помощью платформы потоковых данных, например Amazon Kinesis, с целью извлечения важной информации в режиме реального времени, а затем размещаются в хранилище, например Amazon S3, где преобразуются и загружаются для решения различных задач пакетной обработки.

Возможные проблемы при работе с потоковыми данными

Обработка потоковых данных требует использования двух уровней: уровня хранилища и уровня обработки. Уровень хранилища должен поддерживать очередность записей и строгую непротиворечивость для обеспечения быстрых, экономичных и воспроизводимых операций записи и чтения больших потоков данных. Уровень обработки отвечает за потребление данных, расположенных на уровне хранилища, выполнение вычислений с использованием этих данных и уведомление уровня хранилища о том, какие данные можно удалить за ненадобностью. Кроме того, необходимо предусмотреть масштабируемость, надежность данных и отказоустойчивость как на уровне хранилища, так и на уровне обработки. В результате появилось множество платформ, предоставляющих необходимую инфраструктуру для создания приложений обработки потоковых данных, включая Amazon Kinesis Data Streams, Amazon Kinesis Data Firehose, Amazon Managed Streaming for Apache Kafka (Amazon MSK), Apache Flume, Apache Spark Streaming и Apache Storm.

2.3 Качество данных для тестирования миграции

В нашей повседневной жизни перемещение информации из одного места в другое – не более чем простая операция копирования и вставки. Когда дело доходит до переноса миллионов единиц данных в новую систему, все становится намного сложнее.

Однако многие компании рассматривают даже массовую миграцию данных как низкоуровневую задачу, выполняемую в два клика. Такая первоначальная недооценка приводит к дополнительным расходам времени и денег. Недавние исследования показали, что 55 процентов проектов миграции данных превышали бюджет, а 62 про-

цента оказались сложнее, чем ожидалось, или фактически потерпели неудачу.

Как не попасть в ту же ловушку? Ответ заключается в понимании сути процесса миграции данных, от его триггеров до заключительных этапов.

Что такое миграция данных?

В общих чертах, миграция данных – это перенос существующих исторических данных в новое хранилище, систему или формат файла. Этот процесс не так прост, как может показаться. Он включает в себя множество подготовительных и пост-миграционных мероприятий, включая планирование, создание резервных копий, тестирование качества и проверку результатов. Миграция завершается только при выключении старой системы, базы данных или среды.

Что заставляет компании переносить свои информационные активы. Обычно перенос данных является частью более крупного проекта, такого как

- модернизация или замена устаревшего программного обеспечения ,
- расширение системы и емкости хранения,
- внедрение дополнительной системы, работающей параллельно с существующим приложением,
- переход к централизованной базе данных для устранения разрозненности данных и обеспечения взаимодействия ,
- перенос ИТ-инфраструктуры в облако, или
- слияния и поглощения (M&A), когда ИТ-ландшафты должны быть объединены в единую систему.

Иногда миграцию данных путают с другими процессами, связанными с массовым перемещением данных. Прежде всего, важно прояснить различия между миграцией данных, интеграцией данных и репликацией данных.

Миграция данных против интеграции данных

В отличие от миграции, связанной с внутренней информацией компании, интеграция заключается в объединении данных из нескольких источников вне и внутри компании в единое представление. Это важный элемент стратегии управления данными, который обеспечивает связь между системами и дает доступ к контенту по широкому кругу вопросов. Консолидированные наборы данных – необхо-

димое условие для точного анализа, извлечения бизнес-информации и отчетности.

Миграция данных – это односторонний путь, который заканчивается после того, как вся информация будет доставлена в целевое место. Интеграция, напротив, может быть непрерывным процессом, который включает потоковую передачу данных в реальном времени и обмен информацией между системами.

Миграция данных против репликации данных

При миграции данных после того, как данные полностью перенесены в новое место, в конечном итоге отказываются от старой системы или базы данных. При репликации периодически переносятся данные в целевое расположение, не удаляя и не отбрасывая их источник. Итак, у него есть отправная точка, но нет определенного времени завершения.

Репликация данных может быть частью процесса интеграции данных. Также это может превратиться в миграцию данных – при условии, что исходное хранилище выведено из эксплуатации.

Основные типы миграции данных

Существует шесть обычно используемых типов миграции данных. Однако это разделение не строгое. Конкретный случай передачи данных может относиться, например, к миграции как базы данных, так и в облако, или включать миграцию приложения и базы данных одновременно.

1. Миграция хранилища.

Миграция системы хранения происходит, когда бизнес приобретает современные технологии, отказываясь от устаревшего оборудования. Это влечет за собой перенос данных с одного физического носителя на другой или с физического на виртуальную среду.

Примеры таких миграций: перенос данных

– от бумажных до цифровых документов,

– от жестких дисков (HDD) до более быстрых и надежных твердотельных накопителей (SSD) или

– от мэйнфреймов до облачного хранилища.

Основная причина этого перехода – насущная потребность в модернизации технологий, а не недостаток места для хранения. Когда дело доходит до крупномасштабных систем, процесс миграции может занять годы. Скажем, Sabre, вторая по величине глобальная распреде-

лительная система (GDS), уже более десяти лет переносит свое программное обеспечение и данные с мэйнфреймов на виртуальные серверы. Ожидается, что его миграционный период будет полностью завершен в 2023 г.

2. Перенос базы данных.

База данных – это не просто место для хранения данных. Она обеспечивает структуру для организации информации определенным образом и обычно управляется через систему управления базами данных (СУБД). Итак, в большинстве случаев миграция базы данных означает

– обновление до последней версии СУБД (т.н. однородная миграция),

– переход на новую СУБД от другого провайдера – например, с MySQL на PostgreSQL или с Oracle на MSSQL (так называемая гетерогенная миграция).

Второй случай сложнее первого, особенно если целевая и исходная базы данных поддерживают разные структуры данных. Это делает задачу еще более сложной, когда нужно перемещать данные из устаревших баз данных, таких как Adabas, IMS или IDMS.

3. Перенос приложений.

Когда компания меняет поставщика корпоративного программного обеспечения – например, отель внедряет новую систему управления имуществом или больница заменяет свою устаревшую систему EHR – это требует перемещения данных из одной вычислительной среды в другую. Ключевой проблемой здесь является то, что старая и новая инфраструктуры могут иметь уникальные модели данных и работать с разными форматами данных.

4. Миграция центра обработки данных.

Дата-центр – это физическая инфраструктура, используемая организациями для хранения своих критически важных приложений и данных. Точнее, это очень темная комната с серверами, сетями, коммутаторами и другим ИТ-оборудованием. Таким образом, миграция центра обработки данных может означать разные вещи: от перемещения существующих компьютеров и проводов в другие помещения до перемещения всех цифровых активов, включая данные и бизнес-приложения, на новые серверы и хранилища.

5. Миграция бизнес-процессов.

Этот тип миграции вызван слияниями и поглощениями, оптимизацией бизнеса или реорганизацией для решения конкурентных проблем или выхода на новые рынки. Все эти изменения могут потребовать переноса бизнес-приложений и баз данных с данными о клиентах, продуктах и операциях в новую среду.

6. Миграция в облако.

Миграция в облако – популярный термин, охватывающий все вышеупомянутые случаи, если они связаны с перемещением данных из локальной среды в облако или между различными облачными средами. Gartner ожидает, что к 2024 году облако привлечет более 45 процентов ИТ-расходов и будет доминировать над постоянно растущим числом ИТ-решений.

В зависимости от объемов данных и различий между исходным и целевым местоположениями миграция может занять от 30 минут до месяцев и даже лет. Сложность проекта и стоимость простоя определяют, как именно развернуть процесс.

Подходы к миграции данных

Выбор правильного подхода к миграции – это первый шаг к обеспечению бесперебойной работы проекта без серьезных задержек.

1. Миграция данных большого взрыва.

Преимущества: менее затратный, менее сложный, занимает меньше времени, все изменения происходят один раз.

Недостатки: высокий риск дорогостоящего отказа, требует простоя.

В сценарии большого взрыва перемещаются все активы данных из исходной среды в целевую за одну операцию в относительно короткий промежуток времени.

Системы не работают и недоступны для пользователей, пока данные перемещаются и претерпевают преобразования для соответствия требованиям целевой инфраструктуры. Перенос обычно выполняется во время официальных праздников или выходных, когда клиенты предположительно не используют приложение.

Подход «большого взрыва» позволяет выполнить миграцию в кратчайшие сроки и избавляет от хлопот одновременной работы в старой и новой системах. Однако в эпоху больших данных даже компании среднего размера накапливают огромные объемы информации,

в то время как пропускная способность сетей и шлюзов API не бесконечна. Это ограничение необходимо учитывать с самого начала.

Вердикт. Подход большого взрыва подходит для небольших компаний или предприятий, работающих с небольшими объемами данных. Это не работает для критически важных приложений, которые должны быть доступны 24/7.

2. Тонкая миграция данных.

Преимущества: меньшая подверженность неожиданным сбоям, нулевое время простоя.

Недостатки: дороже, требует больше времени, требует дополнительных усилий и ресурсов для поддержания работы двух систем.

Этот подход, также известный как поэтапная или итеративная миграция, привносит опыт гибкой разработки в перенос данных. Он разбивает весь процесс на подмиграции, каждая со своими целями, сроками, масштабом и проверками качества.

Капельная миграция предполагает параллельную работу старой и новой систем и передачу данных небольшими приращениями. В результате получается преимущество нулевого времени простоя, а клиенты довольны доступностью приложений 24/7.

С другой стороны, итеративная стратегия занимает гораздо больше времени и усложняет проект. Группа по миграции должна отслеживать, какие данные уже были перенесены, и гарантировать, что пользователи могут переключаться между двумя системами для доступа к необходимой информации.

Еще один способ выполнить постепенную миграцию – оставить старое приложение полностью работоспособным до конца миграции. В результате клиенты будут использовать старую систему как обычно и переключатся на новое приложение только после того, как все данные будут успешно загружены в целевую среду.

Однако этот сценарий не облегчает жизнь инженерам. Они должны обеспечить синхронизацию данных в режиме реального времени на двух платформах после их создания или изменения. Другими словами, любые изменения в исходной системе должны запускать обновления в целевой системе.

Вердикт. Капельная миграция – правильный выбор для средних и крупных предприятий, которые не могут себе позволить длительные

простои, но обладают достаточным опытом для решения технологических проблем.

Процесс миграции данных

Независимо от подхода, проект миграции данных проходит одни и те же ключевые фазы, а именно:

– планирование;
– аудит и профилирование данных;
– резервное копирование данных;
– дизайн миграции;
– казнь;
– тестирование;
– постмиграционный аудит.

1. Планирование: необходимо составить план переноса данных и придерживайтесь его.

Миграция данных – сложный процесс, который начинается с оценки существующих активов данных и тщательного составления плана миграции. Этап планирования можно разделить на четыре этапа.

Шаг 1 – уточните объем. Ключевая цель этого шага – отфильтровать любые избыточные данные и определить минимальный объем информации, необходимый для эффективной работы системы. Таким образом, необходимо провести высокоуровневый анализ исходной и целевой систем, проконсультировавшись с пользователями данных, на которых непосредственно повлияют предстоящие изменения.

Шаг 2 – оцените исходную и целевую системы. План перехода должен включать тщательную оценку операционных требований текущей системы и способов их адаптации к новой среде.

Шаг 3 – установите стандарты данных. Это позволит команде выявлять проблемные области на каждом этапе процесса миграции и избегать неожиданных проблем на этапе после миграции.

Шаг 4 – оцените бюджет и установите реалистичные сроки. После уточнения объема работ и оценки систем легче выбрать подход (большой взрыв или тонкая струйка), оценить ресурсы, необходимые для проекта, установить графики и сроки. Согласно Oracle оценок, предприятие масштаб миграции данных проект длится от шести месяцев до двух лет в среднем.

2. Аудит и профилирование данных: используйте цифровые инструменты.

Этот этап предназначен для изучения и очистки всего объема переносимых данных. Он направлен на обнаружение возможных конфликтов, выявление проблем с качеством данных и устранение дублирования и аномалий до миграции.

Аудит и профилирование – утомительная, трудоемкая и трудоемкая деятельность, поэтому в крупных проектах следует использовать инструменты автоматизации. Среди популярных решений – Open Studio for Data Quality, Data Ladder, SAS Data Quality, Informatica Data Quality и IBM InfoSphere QualityStage, и это лишь некоторые из них.

3. Резервное копирование данных: защитите свой контент перед его перемещением.

Технически этот этап не является обязательным. Однако передовой опыт миграции данных диктует создание полной резервной копии содержимого, которое вы планируете переместить, перед выполнением фактической миграции. В результате получается дополнительный уровень защиты в случае непредвиденных сбоев миграции и потери данных.

4. Дизайн миграции.

Схема миграции определяет правила миграции и тестирования, разъясняет критерии приемки и распределяет роли и обязанности между членами группы миграции.

Хотя для переноса данных можно использовать несколько технологий, предпочтительным является извлечение, преобразование и загрузка (ETL). Имеет смысл нанять разработчика ETL или специального инженера-программиста с глубокими знаниями в процессах ETL, особенно если проект имеет дело с большими объемами данных и сложным потоком данных.

На этом этапе разработчики ETL или инженеры по обработке данных создают сценарии для переноса данных или выбирают и настраивают сторонние инструменты ETL. Неотъемлемой частью ETL является отображение данных. В идеальном сценарии он включает в себя не только разработчика ETL, но и системного аналитика, знающего как исходную, так и целевую систему, и бизнес-аналитика, который понимает ценность перемещаемых данных.

Продолжительность этого этапа зависит главным образом от времени, необходимого для написания сценариев для процедур ETL

или приобретения соответствующих средств автоматизации. Если все необходимое программное обеспечение установлено и нужно только настроить его, процесс миграции займет несколько недель. В противном случае это может занять несколько месяцев.

5. Исполнение: фокус на бизнес-целях и удовлетворенности клиентов.

Это когда на самом деле происходит миграция – или извлечение, преобразование и загрузка данных. В сценарии большого взрыва это продлится не более пары дней. В качестве альтернативы, если данные передаются тонкими струйками, выполнение займет гораздо больше времени, но с нулевым временем простоя и минимально возможным риском критических сбоев.

Если был выбран поэтапный подход, необходимо убедиться, что действия по миграции не мешают обычным системным операциям. Кроме того, группа по миграции должна связаться с бизнес-подразделениями, чтобы уточнить, когда следует развертывать каждую частичную миграцию и для какой группы пользователей.

6. Тестирование миграции данных: проверка качества данных на разных этапах.

Фактически, тестирование не является отдельным этапом, поскольку оно проводится на этапах проектирования, выполнения и после миграции. Если выбран капельный подход, то следует протестировать каждую часть перенесенных данных, чтобы своевременно устранить проблемы.

Частое тестирование обеспечивает безопасную передачу элементов данных, их высокое качество и соответствие требованиям при входе в целевую инфраструктуру. Вы можете узнать больше о деталях тестирования процесса ETL из нашей специальной статьи.

7. Пост-миграционный аудит: подтверждение результатов с ключевыми клиентами.

Перед запуском перенесенных данных в производственную среду результаты должны быть подтверждены ключевыми бизнес-пользователями. Этот этап обеспечивает правильную транспортировку и регистрацию информации. После пост-миграционного аудита старую систему можно списать.

Золотые правила миграции данных

Хотя каждый проект миграции данных уникален и сопряжен со своими проблемами, некоторые общие золотые правила могут помочь компаниям безопасно перемещать свои ценные информационные активы, избегая критических задержек:

– Используйте миграцию данных как возможность выявить и исправить проблемы с качеством данных. Установите высокие стандарты для улучшения данных и метаданных при их переносе.

– Нанять специалистов по миграции данных и назначить специальную команду по миграции для запуска проекта.

– Сведите к минимуму объем переносимых данных.

– Профилируйте все исходные данные перед написанием сценариев сопоставления.

– Выделите значительное время на этапе проектирования, так как это имеет большое значение для успеха проекта.

– Не торопитесь выключать старую платформу. Иногда первая попытка переноса данных терпит неудачу, требуя отката и еще одной попытки.

Миграция данных часто рассматривается как неизбежное зло, а не как процесс добавления ценности. И это, кажется, корень многих, если не всех трудностей. Рассмотрение миграции как важного инновационного проекта, заслуживающего особого внимания, – это половина дела.

2.4 Принятие решения о тестовой среде и тестовых данных

Почему тестирование данных важно

Данные играют все большую роль в принятии решений как в повседневной жизни, так и в бизнесе. Современные технологии и алгоритмы позволяют обрабатывать и хранить огромные массивы данных, преобразуя их в полезную информацию.

Что это за данные? Например, история браузера, транзакции по карте, точки перемещения какого-то девайса. Они обезличены, но эти данные все равно принадлежат определенному устройству. Если их собрать и обработать, то можно получить довольно интересную информацию о хозяине этого девайса. Например, куда он любит ходить, какой у него пол и возраст. Так постепенно мы «очеловечиваем» устройство и наделяем его какими-то характеристиками.

Потом эту информацию можно использовать для таргетированной рекламы. Качество таргетирования рекламы может быть повышено благодаря тому, что известно о девайсах, на которых ее показывают.

Все это базируется на тех данных, которыми владеют разные компании и люди. Для эффективного использования этих данных нужно, чтобы они были достоверными и было известно, что именно этому аккаунту принадлежат эти транзакции.

Так как данных становится очень много, их хранение требует значительных ресурсов. Очистка данных – отдельная задача, которую надо решать.

Как формируются требования к качеству данных

Тестировщик начинает выяснять, что ему непонятно и что он хотел бы узнать об объекте тестирования. Он, по идее, должен знать, какими должны быть данные. Тестировщик составляет список вопросов и начинает брать «интервью» у заказчика.

Пример требований – если есть список людей, то имя, фамилия и отчество могут повторяться. Но вся совокупность строк повторяться не может. По одной ячейке могут быть допустимы повторения, а по целой строке или по совокупности нескольких ячеек – уже нет. Полного совпадения не должно быть.

Дальше начинает спрашивать о формате данных в определенной ячейке. Например, в номере телефона должно быть 12 цифр, в номере банковской карты – 16. Также может быть критерий, что не всякая последовательность этих знаков является номером банковской карты. Или, что в фамилии могут быть только буквы. Может возникнуть много вопросов по поводу формата данных. Таким образом, выясняем все, что нам нужно знать о предмете тестирования.

Что такое качественные данные

Качественные данные должны обладать несколькими характеристиками.

1. Полнота – в записях нет пропусков, все ячейки должны быть заполненными. Данные должны нести как можно больше информации.

2. Уникальность – среди данных не должно быть одинаковых записей.

3. Достоверность –ради этого все и затевается. Никто не хочет работать с данными, которым нельзя верить. Ячейки таблиц с качественными данными содержат то, что и должны содержать: IP-адрес, номер телефона и т. д.

4. Точность. Если говорить о цифровых данных, то должно быть точное количество знаков. Например, 12 знаков после запятой. Данные должны быть близки к какому-то среднему значению.

5. Согласованность – данные должны сохранять значения, независимо от способа их измерения.

6. Своевременность – данные должны быть актуальны, особенно если они периодически обновляются. Например, каждый месяц количество данных должно увеличиваться. Данные не должны устаревать. Если мы говорим о банковских транзакциях, то нам интересно, чтобы они были, например, за последние полгода.

Уровни тестирования данных

Необходимо сгруппировать данные по так называемым слоям – здесь работает хорошая аналогия с пирамидой тестирования. Это распределение по количеству тестов на разных уровнях приложения.

Unit-слой – это когда тестируется один модуль программы, чаще всего это одна функция или метод. Таких тестов должно быть больше всего. Unit-тест для данных – это когда определяются требования для каждой ячейки. Нет смысла тестировать дальше, если есть ошибки на уровне ячеек. Если, например, в фамилии содержатся цифры, то какой смысл проверять что-то дальше? Возможно, там должны быть буквы, похожие на эти цифры. И тогда нужно все исправлять и проверять следующий уровень, чтобы все было в единственном числе и не было дубликатов, если так сказано в требованиях.

Integration-слой – это когда несколько кусков программы тестируются вместе. Слой API для данных – это когда говорится о всей таблице. Допустим, могут быть дубликаты, но не больше ста штук. Если, например, город-миллионник, то на одной улице не может жить миллион человек. Поэтому если сделать выборку по улице, то количество адресов должно быть десять тысяч или тысяча – это надо определить. А если получился миллион, то с данными что-то не так.

System-слой – это когда вся программа тестируется полностью. В случае с данными этот слой означает, что тестируется вся система. Здесь включается статистика. Например, говорим, что у нас не может

быть больше 30 % мужчин, рожденных после 1985 г. Или мы говорим, что 80 % данных должны быть одного типа.

Для чего нужна тестовая среда

Как известно, в среде разработки создаются новые сервисы и после того, как они готовы, их переносят в продуктивную среду. Но прежде чем решение будет доступно для общего использования, его необходимо проверить на работоспособность. Для этих целей создаются тестовые среды, которые выступают своего рода полигоном, где проводятся всевозможные проверки. Важно понимать, что процесс тестирования требует системного подхода и использовать для этого лучшие практики недостаточно. Чтобы получить положительные результаты, необходимо с самого начала организовать процесс правильно. Для начала необходимо определиться с целями, областью, методологией тестирования и позаботиться о подготовке тестовой среды.

Сегодня многие организации, в том числе команды разработки, уходят от традиционных подходов к организации тестовых сред. Если раньше под эти задачи разворачивали собственную инфраструктуру, которая требовала отдельной поддержки и дополнительных финансовых вливаний, теперь чаще отдается предпочтение более экономичным вариантам. Одним из них является виртуальный хостинг, представляющий собой один из удобных вариантов организации процесса тестирования, который отличается целым рядом конкурентных преимуществ. Примечательно, что тестовые среды, развернутые на базе виртуального хостинга, избавляют от простоев собственных серверов, поскольку отпадает необходимость в их использовании. Вместо этого вы получаете необходимые виртуальные ресурсы без потери качества.

В последнее время большинство приложений создаются с возможностью доступа через стандартный интернет-браузер, и они также нуждаются в проверке работоспособности на уровне кода. Очень важно, чтобы приложение точно повторяло взаимодействие с пользователем. Кроме того, важно получить обратную связь относительно производительности и надежности работы сервиса. Используя для этих задач виртуальный хостинг, можно получить практически мгновенный ответ и обратную связь относительно функциональности и согласованности работы конкретного решения.

Предположим, необходимо организовать среду для тестирования нового приложения. Для решения этой задачи достаточно обратиться к облачному провайдеру и запросить ресурсы для организации тестовой среды. После клиенту предоставляется доступ к консоли управления, выделяются требуемые ресурсы, которые используются при развертывании тестируемого решения. А после окончания процессов проверки и выявления багов, используемые мощности могут быть возвращены и использоваться для других задач. Такой подход облегчает совместное использование ресурсов и предотвращает простои недостаточно загруженного оборудования.

Виртуальный хостинг также отлично подойдет для проведения нагрузочного тестирования веб-приложений и демонстрационных целей. Поскольку здесь отсутствует необходимость использовать инфраструктуру компании, тестирование происходит без оказания влияния на собственные мощности. За счет возможности потребления выделенных ресурсов, можно получить не приблизительные, а точные результаты.

Тестовая среда на базе виртуального хостинга отлично подойдет для временных проектов. Зачастую возникают ситуации, когда требуется развернуть веб-приложение, проверить его функциональность и в случае корректной работы, передать заказчику. В таком сценарии отсутствует необходимость в использовании виртуального хостинга на постоянной основе. После выявления и исправления багов, тестовая среда деактивируется и используется в дальнейшем по мере необходимости.

Тестовые среды, организованные на базе виртуального хостинга, превращают затраты в минимальные и значительно ускоряют циклы тестирования. Такую среду можно развернуть в кратчайшие сроки, подобрав под себя оптимальную конфигурацию. В результате клиент получает более эффективное и относительно недорогое решение, благодаря которому реализуются проекты по тестированию и разработке с минимальными издержками.

Виды тестовых сред

Тип тестовой среды, которую надо настроить, зависит от продукта и компании. Разберемся в деталях настройки тестирования для различных сценариев:

1. Тестирование на тестовом сервере

Самый простой способ проверить API – это отправить запросы на тестовый сервер с настроенной службой API. QA обычно помогают с доступом к тестовому серверу. На тестовом сервере нужно будет получить соответствующие URL-адреса, идентификаторы входа в систему, роли и т. д. У QA нужно уточнить есть ли что-то, что нельзя изменять или удалять, потому что иногда одна и та же среда тестирования является общей для групп.

Кроме того, нужно убедиться, что логины соответствуют разрешениям, которые будут у пользователей. Если есть логин администратора, ответы могут отличаться от пользовательских.

Возможно, понадобиться создание определенных файлов, необходимых для настройки сервера с параметрами, которые нужно проверить. Понимание того, как именно создавать файлы, каталоги для их загрузки, службы для остановки и перезапуска и т.д., может потребовать дополнительных исследований.

Что именно нужно сделать, зависит от продукта, среды, компании, ограничений безопасности и т. д. Нет двух одинаковых компаний. Иногда настроить тестовую систему сложно, а иногда – просто.

В одной компании, для получения доступа к тестовой системе, может понадобиться пройти ряд препятствий безопасности. Например, для подключения к веб-сервисам из внутренних систем разработчики должны пройти через промежуточный сервер. Чтобы подключиться к тестовому экземпляру веб-сервера, нужно подключиться к серверу-посреднику по протоколу SSL, а затем подключиться к серверу-посреднику. (Это не то, что придется делать пользователям, эти прелести только для внутренних инженеров.)

2. Тестирование локальных сборок.

Часто разработчики работают с локальным экземпляром системы, прежде чем отправить его на тестовый сервер. Другими словами, они создают приложение или веб-сервер полностью на своих компьютерах и запускают там тестовый код задолго до отправки его на тестовый сервер. Если подключиться к проекту на этом этапе – замечательно, можно больше влиять на дизайн API и влиять на изменения по мере необходимости. Чтобы создать код локально, может потребоваться установить специальные утилиты или платформы, ознакомиться с различными операциями командной строки для создания кода и многое другое.

Имеет смысл обладать возможностью запуска локальных сборок на своем собственном компьютере, поскольку дает возможность документировать контент заранее, задолго до выпуска.

При локальных сборках настройка функциональной системы намного сложнее, чем использование тестового сервера. Тем не менее, при желании написать хорошую документацию, необходимо настроить тестовую систему. Хорошие разработчики знают и осознают эту потребность, и поэтому они обычно (в некоторой степени) помогают настроить тестовую среду для начала работы.

3. Тестирование примеров приложений.

В зависимости от продукта, возможно понадобится пример кода приложения или SDK в результатах кода. Пример приложения или SDK (или несколько приложений и SDK на разных языках программирования) часто идет с продуктом, чтобы продемонстрировать, как интегрировать и вызывать API. При наличии тестового приложения, которое интегрирует API, вероятно, потребуется установить некоторые программы или платформы на свой компьютер, чтобы получить пример приложения.

Например, для создания примера Java-приложения для взаимодействия с системой, понадобится установить Java Development Kit и Java IDE на компьютер, чтобы приложение заработало. Если приложение на PHP – нужно установить PHP. Или, если это приложение для Android, потребуется загрузить Android Studio и подключить его к виртуальному (или реальному) устройству.

Инструкций по запуску примера приложения обычно мало, поскольку разработчики предполагают, что пользователи уже настроили эти среды на своих компьютерах. (Для пользователя не имеет смысла выбирать приложение Java, если у него, например, еще не было среды Java.)

Образец приложения является одной из самых полезных частей документов. Когда пример приложения будет настроен и заработает, нужно найти возможность добавления документации в комментарии к коду. По крайней мере, привести пример приложения к работе на своем компьютере и включить шаги по настройке в свою документацию.

4. Тестирование аппаратных продуктов.

При документировании аппаратного продукта, нужно защитить устройство, на котором установлена правильная сборка. Крупные компании часто имеют в наличии прототип устройства. В некоторых

компаниях могут быть киоски, в которые можно «прошить» (быстро установить) определенный номер сборки на устройстве. Или можно отправить серийный номер устройства управляющему пулом устройств, которые получают обновления бета-версии из облака.

Иногда бывает сложно получить тестовый экземпляр аппаратного продукта для тестирования. Например в правительственном учреждении, при документировании массива хранения на миллион долларов. Может, представиться только один единственный раз, когда разрешат увидеть массив хранилищ, войти в специальную среду серверной в сопровождении инженера, который не мечтал бы позволить по-настоящему прикоснуться к массиву стороннему человеку, а тем более выгрузить диск хранилища и выполнить команды в терминале, заменить RAID или сделать еще какую-нибудь другую задачу (для которой пишется инструкция).

При создании документации на оборудование, необходим доступ к нему, чтобы предоставить надежную документацию по его использованию. Нужно понимать, как запускать приложения на устройстве и как взаимодействовать с ним.

2.5 Параметры качества данных

Данные – это фундамент, на котором держится компания с управлением на основе данных.

Если люди, принимающие решения, не располагают своевременной, релевантной и достоверной информацией, у них не остается другого выхода, как только положиться на собственную интуицию. Качество данных – ключевой аспект.

Специалистам аналитикам нужны правильные данные, собранные правильным образом и в правильной форме, в правильном месте, в правильное время. (Они просят совсем не много.) Если какое то из этих требований не выполнено или выполнено недостаточно хорошо, у аналитиков сужается круг вопросов, на которые они способны дать ответ, а также снижается качество выводов, которые они могут сделать на основании данных.

Активная проверка и сохранение качества данных – совместная обязанность всех сотрудников. Каждый участник аналитической цепочки ценности должен следить за качеством данных. Таким образом,

каждому участнику будет полезно на более глубоком уровне разбираться в этом вопросе.

Аспекты качества данных

Качество данных невозможно свести к одной цифре. Качество – это не 5 или 32. Причина в том, что это понятие охватывает целый ряд аспектов, или направлений. Соответственно, начинают выделять уровни качества, при которых одни аспекты оказываются более серьезными, чем другие. Важность этих аспектов зависит от контекста анализа, который должен быть выполнен с этими данными.

Итак, качество данных определяется несколькими аспектами. Данные должны отвечать ряду требований.

1. Доступность.

У аналитика должен быть доступ к данным. Это предполагает не только разрешение на их получение, но также наличие соответствующих инструментов, обеспечивающих возможность их использовать и анализировать. Например, в файле дампа памяти SQL (Structured Query Language – языка структурированных запросов при работе с базой данных) содержится информация, которая может потребоваться аналитику, но не в той форме, в которой он сможет ее использовать. Для работы с этими данными они должны быть представлены в работающей базе данных или в инструментах бизнес□ аналитики(подключенных к этой базе данных).

2. Точность.

Данные должны отражать истинные значения или положение дел. Например, показания неправильно настроенного термометра, ошибка в дате рождения или устаревший адрес – это все примеры неточных данных.

3. Взаимосвязанность.

Должна быть возможность точно связать одни данные с другими. Например, заказ клиента должен быть связан с информацией о нем самом, с товаром или товарами из заказа, с платежной информацией и информацией об адресе доставки. Этот набор данных обеспечивает полную картину заказа клиента. Взаимосвязь обеспечивается набором идентификационных кодов или ключей, связывающих воедино информацию из разных частей базы данных.

4. Полнота.

Под неполными данными может подразумеваться как отсутствие части информации (например, в сведениях о клиенте не указано его имя), так и полное отсутствие единицы информации (например, в результате ошибки при сохранении в базу данных потерялась вся информация о клиенте).

5. Непротиворечивость.

Данные должны быть согласованными. Например, адрес конкретного клиента в одной базе данных должен совпадать с адресом этого же клиента в другой базе. При наличии разногласий один из источников следует считать основным или вообще не использовать сомнительные данные до устранения причины разногласий.

6. Однозначность.

Каждое поле, содержащее индивидуальные данные, имеет определенное, недвусмысленное значение. Четко названные поля в совокупности со словарем базы данных (подробнее об этом чуть позже) помогают обеспечить качество данных.

7. Релевантность.

Данные зависят от характера анализа. Например, исторический экскурс по биржевым ценам Американской ассоциации землевладельцев может быть интересным, но при этом не иметь никакого отношения к анализу фьючерсных контрактов на грудинную свинину.

8. Надежность.

Данные должны быть одновременно полными (то есть содержать все сведения, которые вы ожидали получить) и точными (то есть отражать достоверную информацию).

9. Своевременность.

Между сбором данных и их доступностью для использования в аналитической работе всегда проходит время. На практике это означает, что аналитики получают данные как раз вовремя, чтобы завершить анализ к необходимому сроку. Недавно мне довелось узнать об одной крупной корпорации, у которой время ожидания при работе с хранилищем данных составляет до одного месяца. При такой задержке данные становятся практически бесполезными (при сохранении издержек на их хранение и обработку), их можно использовать только в целях долгосрочного стратегического планирования и прогнозирования.

Ошибка всего в одном из этих аспектов может привести к тому, что данные окажутся частично или полностью непригодными к ис-

пользованию или, хуже того, будут казаться достоверными, но приведут к неправильным выводам.

Данные с ошибками

Ошибки могут появиться в данных по многим причинам и на любом этапе сбора информации. Давайте проследим весь жизненный цикл данных с момента их генерации и до момента анализа и посмотрим, как на каждом из этапов в данные могут закрадываться ошибки.

В данных всегда больше ошибок, чем кажется. По результатам одного из исследований, ежегодно американские компании терпят ущерб почти в 600 млн долл. из-за ошибочных данных или данных плохого качества (это 3,5 % ВВП!).

Во многих случаях аналитики лишены возможности контролировать сбор и первичную обработку данных. Обычно они бывают одним из последних звеньев в длинной цепочке по генерации данных, их фиксированию, передаче, обработке и объединению. Тем не менее важно понимать, какие проблемы с качеством данных могут возникнуть и как их потенциально можно разрешить.

Наиболее распространенные причины ошибок в данных:

1. Генерация данных.
2. Ввод данных.
3. Дублирование данных.
4. Усеченные данные.
5. Несовпадение единиц измерения.
6. Значения по умолчанию.

Показатели качества публичных данных

Проблема качества данных представляет собой достаточно серьезную тему и не только в связи с их обработкой и анализом. На данных в современном цифровом мире построено множество процессов, в том числе и связанных с безопасностью. Поэтому от того, насколько качественные данные используются в государственных и коммерческих организациях зависит эффективность и результат их работы.

Рассмотрим несколько показателей, которые могли бы составить интегрированную оценку качества публичных (открытых) данных.

1. Актуальность данных.

Обозначенный или косвенно определяемый момент времени, на который данные отражают реальное состояние целевого субъекта (объекта, системы, явления, модели, события и т.п.).

Актуальность данных также может быть обозначена через период времени в течение которого они сохраняют свою значимость. Учитывая постоянные изменения экономических систем, публичные экономические данные имеют достаточно короткие сроки актуальности.

Актуальность данных чаще всего устанавливается поставщиком, в дополнение к которой он также может «дать обещание» периодического их обновления для ее поддержания.

Получатель данных может самостоятельно оценивать их актуальность на основании информации от поставщика или иными способами.

2. Объективность данных.

Точность отражения данными реального состояния целевого субъекта (объекта, системы, явления, модели, события и т.п.).

Объективность напрямую зависит от применяемого метода и процедур сбора информации, а также от плотности регистрируемых данных. В процессе обработки наборов цифровых данных, они теряют свою объективность и обогащаются агрегированными, округленными, приведенными и расчетными показателями. Однако за счет этого данные «насыщаются» знаниями, тем самым позволяя в последующем сокращать последовательность операций по извлечению из них значимых для практики сведений.

Поставщик может указать объективность публичных данных охарактеризовав их первичность и описав процедуру их получения.

Получатель вправе критично отнестись к вторичным данным, особенно если их объективность не доказана применяемыми формулами и математическими расчетными моделями.

3. Целостность данных.

Полнота отражения данными реального состояния целевого субъекта (объекта, системы, явления, модели, события и т. п.).

В отличии от объективности, целостность показывает насколько полными и безошибочными являются данные как в части смыслового непротиворечия, так и в части соответствия заданной структуре или выбранного формата. Целостность зависит от корректного разделения

на элементарные неделимые единицы, сохранения их неделимости, правильной идентификации и взаимной связанности.

Данные публикуемые добросовестным поставщиком по умолчанию должны являться целостными.

Получатель определяет целостность специальными проверочными методами оценивая смысловое содержание, корректность определения структуры и технически проверяя формат.

4. Релевантность данных.

Соответствие данных о реальном состоянии целевого субъекта (объекта, системы, явления, модели, события и т.п.) решаемой задачи (поставленной цели) и возможность их применения с учетом имеющегося содержания, структуры и формата.

Понимание релевантности напрямую увязывается с целью пользователя данных и конкретной исполняемой им задачи, а значит и с располагаемым исходным набором данных.

Поставщик не может повлиять на релевантность данных, но может существенно упростить понимание данного показателя качества с помощью расширенных метаданных, применения распространенных форматов и традиционных структур, а также указанием рекомендаций по их использованию.

Получатель в каждом конкретном случае оценивает релевантность наборов данных исходя из тематики и рабочего формата (т.е. используемых инструментов).

5. Совместимость данных.

Совместная обработка данных о реальном состоянии целевого субъекта (объекта, системы, явления, модели, события и т.п.) с имеющимися в рамках решаемой задачи (поставленной цели).

В отличие от релевантности, совместимость – это процедурный показатель, который характеризует возможность включить данные в обрабатываемый массив для дальнейшего анализа и не связан напрямую с сутью и критериями текущей задачи. С другой стороны, совместимость на содержательном уровне с тематикой исполняемой задачи важна для эффективной обработки цифровых данных. Публичные данные должны особенно тщательно оцениваться на совместимость, в том числе с точки зрения их разновидности. Допустимо ли для конкретных целей совмещение – взаимное использование – открытых данных и разделяемых данных или разделяемы и делегируемых данных зависит от оценки аналитика. Чаще всего необходимо

соблюдать условия раздельного хранения и контроля разных видов публичных данных.

Поставщик публичных данных задает совместимость через метаданные и ссылки на контекст.

Получатель определяет возможность совместного использования данных для каждого набора как по содержанию и структуре, так и по формату. Но в отличие от релевантности, несовместимые данные можно попытаться привести к совместимому с помощью различных операций трансформации, перекодирования, перевода и т. п.

6. Измеримость данных.

Присутствие в данных обрабатываемых качественных или количественных характеристик реального состояния целевого субъекта (объекта, системы, явления, модели, события и т.п.), а также подсчитанный конечный объем набора цифровых данных.

Содержательная измеримость данных является основой для выполнения последующих процедур их обработки и анализа. Измерение же общего объема данных необходимо для выбора инструментария и контроля их целостности в процессе обработки и по итогам анализа.

Поставщик может явно указывать «измерения», включенные в данные, как количественные, так и качественные. Как минимум, сопровождение наборов публичных данных записью об итоговом или пофайловом их размере в байтах почти является общепринятым стандартом.

Получатель публичных данных восстанавливает измеримость в содержании данных анализируя их и исследуя структуру и всегда точно или бегло проверяет насколько их физический размер соответствует заявленному.

7. Управляемость данных.

Возможность целевым и осмысленным образом обработать, передать и контролировать данные о реальном состоянии целевого субъекта (объекта, системы, явления, модели, события и т. п.).

Управляемость обусловлена необходимостью изменять, исправлять, структурировать, организовывать, фильтровать, сохранять, пересылать, оценивать, распределять данные. Она во многом основывается на правильно выбранной структуре и формате.

Поставщик может заявить об управляемости данных через сопровождение их специальными метаданными, но получатель, как

правило, самостоятельно проводит ее оценку исходя из имеющихся у него компетенций и инструментов.

8. Привязка к источнику данных

Связанная и достоверная идентификация цепочки поставки данных о реальном состоянии целевого субъекта (объекта, системы, явления, модели, события и т. п.).

При этом в описание «цепочки поставки публичных данных» лучше включить указания на все субъекты, которые исполняли основные роли трансфера данных: генератор (автор), владелец, поставщик. Привязка к источнику позволяет поставщику и получателю сослаться и восстановить авторство, правоотношения, достоверность источника, доверие к распространителям.

Публичные данные почти всегда распространяются с указанием владельца и поставщика. И более того, одним из ограничений использования данных является необходимость указать первоисточник при их последующей публикации или использовании. Следует учитывать, что хорошая привязка данных позволяет по необходимости получить ее повторно с уточнениями, дополнительной актуализацией или с восстановленной целостностью, т. е. – с повышенным качеством.

9. Доверие к поставщику данных.

Оценка получателем деловых качеств поставщика публичных данных о целевом состоянии субъекта (объекта, системы, явления, модели, события и т.п.), как ответственного, авторитетного, организованного и относительно независимого издателя цифровой информации высокого качества.

Данный показатель выступает некоторой интегрированной ретроспективной оценкой всех предыдущих трансферов данных поставщика – репутация издателя публичных данных.

Получатель всегда исходит из внутренней убежденности при определении такого показателя качества данных, но у поставщика есть несколько путей по формированию и поддержанию нужного ему уровня доверия. К ним можно, например, отнести: тщательную подготовку данных для публичного трансфера, высокий уровень организации процессов издания «цифры», поддержку обратной связи с получателями, своевременную актуализацию и извещение об обнаруженных в данных проблемах, специальные мероприятия, участие в независимой оценке и ассоциациях.

Любой из указанных показателей качества данных субъективен, как в части смыслового содержания данных, так и в части его восприятия разными поставщиками и получателями.

Тем не менее все показатели можно разделить на:

– условно-объективные – это показатели, значения которых слабо зависят от мнения поставщика или получателя данных и устанавливаются в соответствии с контролируемыми и частично проверяемыми критериями, к ним относятся: актуальность, целостность, измеримость, совместимость, привязка к источнику.

– условно-субъективные – это показатели, значения которых напрямую зависят от мнения поставщика или получателя данных и устанавливаются в соответствии с внутренней «убежденностью» как некоторая допустимая критериальная оценка, к ним относятся: объективность, релевантность, управляемость, доверие к поставщику.

Формальная оценка каждого из показателей качества может осуществляться как в баллах (в заданном интервале), так и в процентах. Причем бальная оценка может даваться экспертным путем, а процент может высчитываться как доля данных отвечающих заданному показателю качества к общему объему данных. В последнем случае задача выглядит много более сложная и требует специальных инструментов, хотя и будет давать взвешенную, но все-таки экспертную оценку качества. Одним их важных аспектов формальной оценки показателей качества является их контроль по мере работы с наборами цифровых данных. В динамике качество данных не должно ухудшаться, т. е. экспертная оценка данных не должна неуправляемо снижаться после отдельных операций или целой серии обработок.

Общая проблема качества публичных данных зависит как от каждого из перечисленных показателей, так и от интегрированной субъективной оценки получателя. В любом случае, качество важно в первую очередь получателю, как лицу выполняющему операции обработки и анализа.

В случае завершения обратной связи стороннего результативного пользователя данных с поставщиком, «проблема» качества данных возвращается последнему «бумерангом». Если данные были предоставлены «плохие» или с ошибками, то ожидать от тех, кто их использовал, сколь-либо хороших и адекватных итогов не приходится. Тогда утрачивается весь смысл усилий по выбору, подготовке и публикации

данных – поставщик не получает никаких новых полезных решений и знаний (продуктов или сервисов).

Характеристики качества данных

Управление качеством данных является неотъемлемой частью управления данными.

Качество данных характеризует степень соответствия данных требованиям пользователей данных в части полноты, целостности, точности, актуальности, согласованности, методологической определенности, соответствия форматам и правилам контроля, достоверности и своевременности. Качество данных непосредственно влияет на применимость и эффективность использования данных для решения задач пользователей данных.

Описание характеристик качества данных:

- Согласованность – показатель качества данных, определяющий насколько непротиворечивыми являются данные в различных связанных наборах данных.
- Своевременность – показатель качества данных, определяющий доступность данных пользователю в нужный момент времени.
- Актуальность – показатель качества данных, определяющий степень соответствия данных моделируемой области на определенный момент времени.
- Целостность – показатель качества данных, определяющий наличие корректных ссылок между сущностями и соответствие их установленным правилам и ограничениям между сущностями (связи «один к одному», «один ко многим»).
- Точность – показатель качества данных, определяющий необходимый уровень детализации данных.
- Полнота – показатель качества данных, определяющий достаточность заполнения наборов данных и их атрибутов.
- Достоверность – показатель качества данных, определяющий насколько корректно данные описывают моделируемую область.

2.6 Работа с базами данных NoSql (Cosmos DB)

В последнее время термин “NoSQL” стал очень модным и популярным, активно развиваются и продвигаются всевозможные про-

граммные решения под этой вывеской. Синонимом NoSQL стали огромные объемы данных, линейная масштабируемость, кластеры, отказоустойчивость, нереляционность. Однако, мало у кого есть четкое понимание, что же такое NoSQL хранилища, как появился этот термин и какими общими характеристиками они обладают. Попробуем устранить этот пробел.

Самое интересное в термине, что при том, что впервые он стал использоваться в конце 90-х, реальный смысл в том виде, как он используется сейчас, приобрел только в середине 2009. Изначально так называлась опенсорсная база данных, созданная Карло Строззи, которая хранила все данные как ASCII файлы и использовала шелловские скрипты вместо SQL для доступа к данным. С “NoSQL” в его нынешнем виде она ничего общего не имела.

В июне 2009 г. в Сан-Франциско Йоханом Оскарссоном была организована встреча, на которой планировалось обсудить новые веяния на ИТ рынке хранения и обработки данных. Главным стимулом для встречи стали новые опенсорсные продукты наподобие BigTable и Dynamo. Для яркой вывески для встречи требовалось найти емкий и лаконичный термин, который отлично укладывался бы в Твиттеровский хэштег. Один из таких терминов предложил Эрик Эванс из RackSpace – «NoSQL». Термин планировался лишь на одну встречу и не имел под собой глубокой смысловой нагрузки, но так получилось, что он распространился по мировой сети наподобие вирусной рекламы и стал де-факто названием целого направления в ИТ-индустрии. На конференции, к слову, выступали Voldemort (клон Amazon Dynamo), Cassandra, Hbase (аналоги Google BigTable), Hypertable, CouchDB, MongoDB.

Стоит еще раз подчеркнуть, что термин “NoSQL” имеет абсолютно стихийное происхождение и не имеет общепризнанного определения или научного учреждения за спиной. Это название скорее характеризует вектор развития ИТ в сторону от реляционных баз данных. Расшифровывается как Not Only SQL, хотя есть сторонники и прямого определения No SQL. Сгруппировать и систематизировать знания о NoSQL мире попытались сделать Прамод Садаладж и Мартин Фаулер в своей недавней книге “NoSQL Distilled”.

Характеристики NoSQL баз данных

Общих характеристик для всех NoSQL немного, так как под лэйблом NoSQL сейчас скрывается множество разнородных систем. Многие характеристики свойственны только определенным NoSQL базам.

1. Не используется SQL.

Имеется в виду ANSI SQL DML, так как многие базы пытаются использовать query languages похожие на общеизвестный любимый синтаксис, но полностью его реализовать не удалось никому и вряд ли удастся.

2. Неструктурированные (schemaless).

Смысл таков, что в NoSQL базах в отличие от реляционных структура данных не регламентирована (или слабо типизированна, если проводить аналогии с языками прогаммирования) – в отдельной строке или документе можно добавить произвольное поле без предварительного декларативного изменения структуры всей таблицы. Таким образом, если появляется необходимость поменять модель данных, то единственное достаточное действие – отразить изменение в коде приложения.

Например, при переименовании поля в MongoDB:

```
BasicDBObject order = new BasicDBObject();
order.put(“date”, orderDate); // это поле было давно
order.put(“totalSum”, total); // раньше мы использовали просто “sum”
```

Если меняется логика приложения, значит ожидается новое поле также и при чтении. Но в силу отсутствия схемы данных поле totalSum отсутствует у других уже существующих объектов Order. В этой ситуации есть два варианта дальнейших действий. Первый – обойти все документы и обновить это поле во всех существующих документах. В силу объемов данных этот процесс происходит без каких-либо блокировок (сравним с командой alter table rename column), поэтому во время обновления уже существующие данные могут считываться другими процессами. Поэтому второй вариант – проверка в коде приложения – неизбежен:

```
BasicDBObject order = new BasicDBObject();
Double totalSum = order.getDouble(“sum”); // Это старая модель
if (totalSum  == null)
        totalSum = order.getDouble(“totalSum”); // Это обновленная
```
модель

А уже при повторной записи запишется это поле в базу в новом формате.

Приятное следствие отсутствия схемы – эффективность работы с разреженными (sparse) данными. Если в одном документе есть поле date_published, а во втором – нет, значит никакого пустого поля date_published для второго создано не будет. Это, в принципе, логично, но менее очевидный пример – column-family NoSQL базы данных, в которых используются знакомые понятия таблиц/колонок. Однако в силу отсутствия схемы, колонки не объявляются декларативно и могут меняться/добавляться во время пользовательской сессии работы с базой. Это позволяет в частности использовать динамические колонки для реализации списков.

У неструктурированной схемы есть свои недостатки – помимо упомянутых выше накладных расходов в коде приложения при смене модели данных – отсутствие всевозможных ограничений со стороны базы (not null, unique, check constraint и т.д.), плюс возникают дополнительные сложности в понимании и контроле структуры данных при параллельной работе с базой разных проектов (отсутствуют какие-либо словари на стороне базы). Впрочем, в условиях быстро меняющегося современного мира такая гибкость является все-таки преимуществом. В качестве примера можно привести Твиттер, который лет пять назад вместе с твиттом хранил лишь немного дополнительной информации (время, Twitter handle и еще несколько байтов метаинформации), однако сейчас в дополнение к самому сообщению в базе сохраняется еще несколько килобайт метаданных.

3. Представление данных в виде агрегатов (aggregates).

В отличие от реляционной модели, которая сохраняет логическую бизнес-сущность приложения в различные физические таблицы в целях нормализации, NoSQL хранилища оперируют с этими сущностями как с целостными объектами (рисунок 2.1.)

В этом примере продемонстрированы агрегаты для стандартной концептуальной реляционной модели e-commerce “заказ – позиции

заказа – платежи – продукт". В обоих случаях заказ объединяется с позициями в один логический объект, при этом каждая позиция хранит в себе ссылку на продукт и некоторые его атрибуты, например, название (такая денормализация необходима, чтобы не запрашивать объект продукта при извлечении заказа – главное правило распределенных систем – минимум "джоинов" между объектами). В одном агрегате платежи объединены с заказом и являются составной частью объекта, в другом – вынесены в отдельный объект. Этим демонстрируется главное правило проектирования структуры данных в NoSQL базах – она должна подчиняться требованиям приложения и быть максимально оптимизированной под наиболее частые запросы. Если платежи регулярно извлекаются вместе с заказом – имеет смысл их включать в общий объект, если же многие запросы работают только с платежами – значит, лучше их вынести в отдельную сущность.

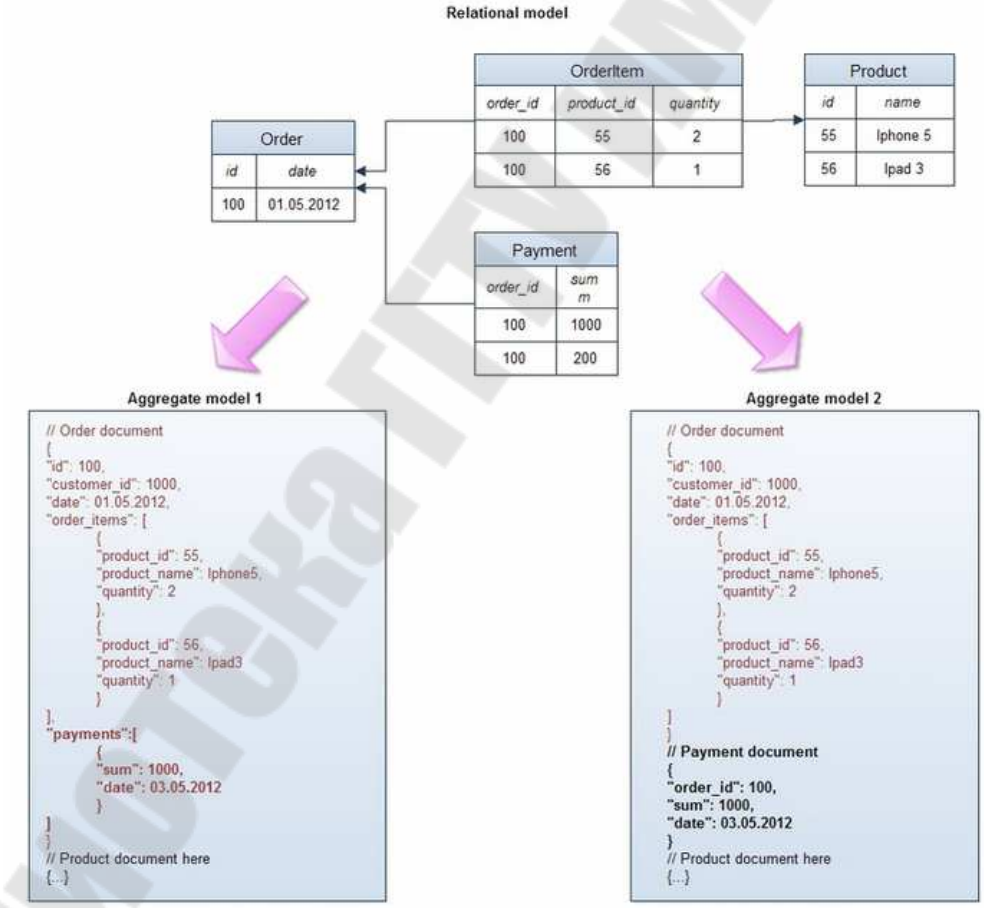

Рисунок 2.1. Пример NoSQL хранилища

В одном агрегате платежи объединены с заказом и являются составной частью объекта, в другом – вынесены в отдельный объект.

Этим демонстрируется главное правило проектирования структуры данных в NoSQL базах – она должна подчиняться требованиям приложения и быть максимально оптимизированной под наиболее частые запросы. Если платежи регулярно извлекаются вместе с заказом – имеет смысл их включать в общий объект, если же многие запросы работают только с платежами – значит, лучше их вынести в отдельную сущность.

Многие возразят, заметив, что работа с большими, часто денормализованными, объектами чревата многочисленными проблемами при попытках произвольных запросов к данным, когда запросы не укладываются в структуру агрегатов. Что, если используются заказы вместе с позициями и платежами по заказу (так работает приложение), но бизнес просит посчитать, сколько единиц определенного продукта было проданно в прошлом месяце? В этом случае вместо сканирования таблицы OrderItem (в случае реляционной модели) придется извлекать заказы целиком в NoSQL хранилище, хотя большая часть этой информации будет не нужна. К сожалению, это компромисс, на который приходится идти в распределенной системе: нет возможности проводить нормализацию данных как в обычной односерверной системе, так как это создаст необходимость объединения данных с разных узлов и может привести к значительному замедлению работы базы

Плюсы и минусы обоих подходов показаны в таблице 2.2.

Таблица 2.2. Плюсы и минусы SQL и NoSQL

Нормализация данных	Данные в виде агрегатов
– Целостность информации при обновлении (меняем запись в одной таблице, а не в нескольких) – Ориентированность на широкий спектр запросов к данным	– Оптимизация только под определенный вид запросов – Сложности при обновлении денормализованных данных
– Неэффективна в распределенной среде – Низкая скорость чтения при использовании объединений – Несоответствие объетной модели приложения физической структуре данных	– Лучший способ добиться большой скорости на чтение в распределенной среде – Возможность хранить физически объекты в том виде, в каком с ними работает приложение – Родная поддержка атомарности на уровне записей

4. Слабые ACID свойства.

Долгое время консистентность (consistency) данных была “священной коровой” для архитекторов и разработчиков. Все реляционные базы обеспечивали тот или иной уровень изоляции – либо за счет блокировок при изменении и блокирующего чтения, либо за счет undo-логов. С приходом огромных массивов информации и распределенных систем стало ясно, что обеспечить для них транзакционность набора операций с одной стороны и получить высокую доступность и быстрое время отклика с другой – невозможно. Более того, даже обновление одной записи не гарантирует, что любой другой пользователь моментально увидит изменения в системе, ведь изменение может произойти, например, в мастер-ноде, а реплика асинхронно скопируется на слейв-ноду, с которой и работает другой пользователь. В таком случае он увидит результат через какой-то промежуток времени. Это называется eventual consistency и это то, на что идут сейчас все крупнейшие интернет-компании мира, включая Facebook и Amazon. Последние с гордостью заявляют, что максимальный интервал, в течение которого пользователь может видеть неконсистентные данные составляют не более секунды. Пример такой ситуации показан на рисунке 2.20.

Рисунок 2.2. Пример одновременного бронирования билетов

Логичный вопрос, который появляется в такой ситуации – а что делать системам, которые классически предъявляют высокие требования к атомарности-консистентности операций и в то же время нуждаются в быстрых распределенных кластерах – финансовым, интер-

нет-магазинам и т. д? Практика показывает, что эти требования уже давно неактуальны: вот что сказал один разработчик финансовой банковской системы: “Если бы мы действительно ждали завершения каждой транзакции в мировой сети ATM (банкоматов), транзакции занимали бы столько времени, что клиенты убегали бы прочь в ярости. Что происходит, если ты и твой партнер снимаете деньги одновременно и превышаете лимит? – Вы оба получите деньги, а мы поправим это позже.” Другой пример – бронирование гостиниц, показанный на картинке. Онлайн-магазины, чья политика работы с данными предполагает eventual consistency, обязаны предусмотреть меры на случай таких ситуаций (автоматическое решение конфликтов, откат операции, обновление с другими данными). На практике гостиницы всегда стараются держать “пул” свободных номеров на непредвиденный случай и это может стать решением спорной ситуации.

На самом деле слабые ACID свойства не означают, что их нет вообще. В большинстве случаев приложение, работающее с реляционной базой данных, использует транзакцию для изменения логически связанных объектов (заказ – позиции заказа), что необходимо, так как это разные таблицы. При правильном проектировании модели данных в NoSQL базе (агрегат представляет из себя заказ вместе с перечнем пунктов заказа) можно добиться такого же самого уровня изоляции при изменении одной записи, что и в реляционной базе данных.

5. Распределенные системы, без совместно используемых ресурсов (share nothing).

Опять же, это не касается граф баз данных, чья структура по определению плохо разносится по удаленным нодам.

Это, возможно, главный лейтмотив развития NoSQL баз. С лавинообразным ростом информации в мире и необходимости ее обрабатывать за разумное время встала проблема вертикальной масштабируемости – рост скорости процессора остановился на 3.5 Ггц, скорость чтения с диска также растет тихими темпами, плюс цена мощного сервера всегда больше суммарной цены нескольких простых серверов. В этой ситуации обычные реляционные базы, даже кластеризованные на массиве дисков, не способны решить проблему скорости, масштабируемости и пропускной способности. Единственный выход из ситуации – горизонтальное масштабирование, когда несколько независимых серверов соединяются быстрой сетью и каждый владеет/обрабатывает только часть данных и/или только часть запро-

сов на чтение-обновление. В такой архитектуре для повышения мощности хранилища (емкости, времени отклика, пропускной способности) необходимо лишь добавить новый сервер в кластер – и все. Процедурами шардинга, репликации, обеспечением отказоустойчивости, перераспределения данных в случае добавления ноды занимается сама NoSQL база. Вкратце представлю основные свойства распределенных NoSQL баз:

Репликация – копирование данных на другие узлы при обновлении. Позволяет как добиться большей масштабируемости, так и повысить доступность и сохранность данных. Принято подразделять на два вида:

- master-slave (рисунок 2.3).

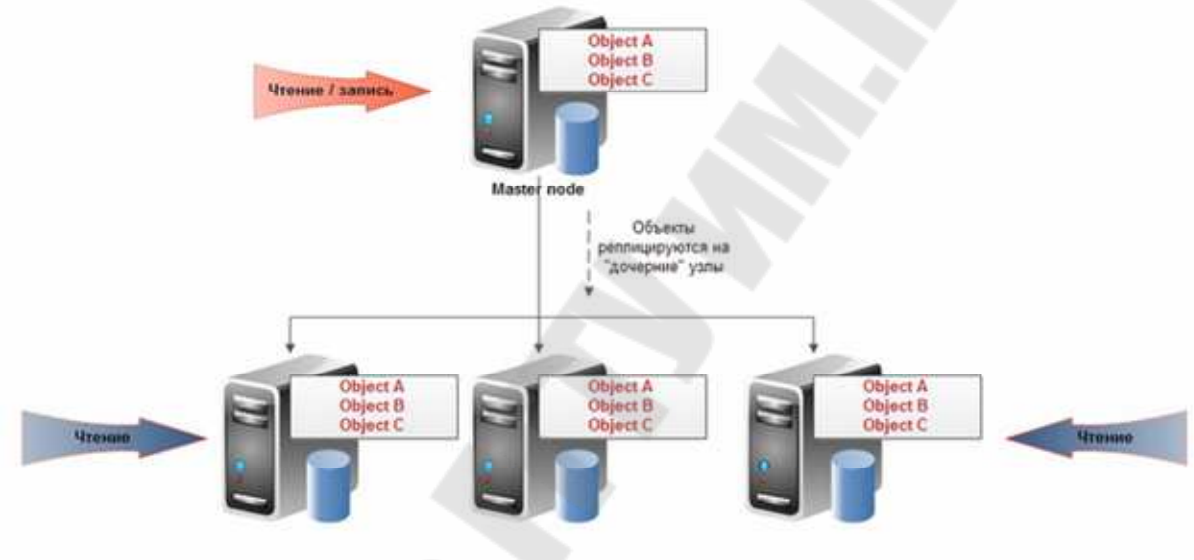

Рисунок 2.3. Master-slave

- и peer-to-peer (рисунок 2.4)

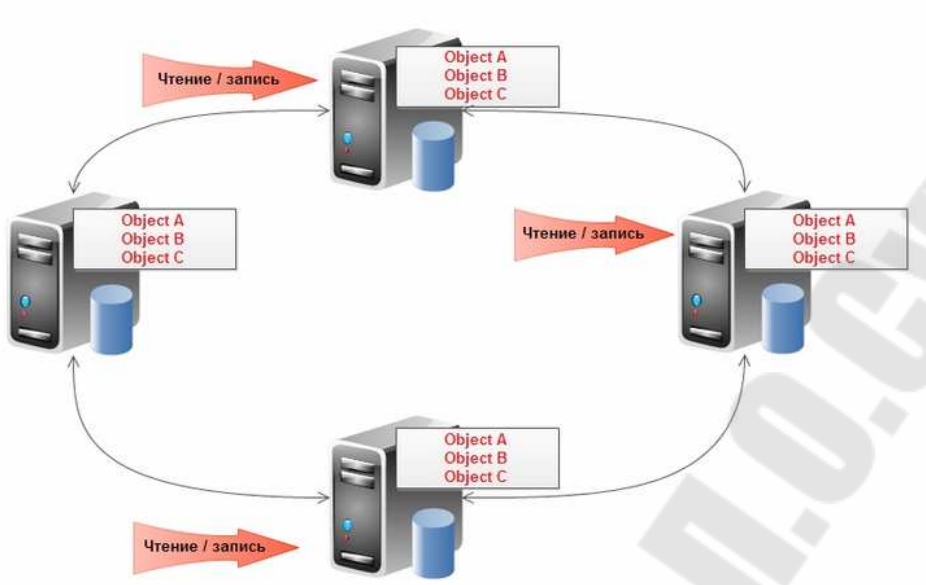

Рисунок 2.4. Peer-to-peer

Первый тип предполагает хорошую масштабируемость на чтение (может происходить с любого узла), но немасштабируемую запись (только в мастер узел). Также есть тонкости с обеспечением постоянной доступности (в случае падения мастера либо вручную, либо автоматически на его место назначается один из оставшихся узлов). Для второго типа репликации предполагается, что все узлы равны и могут обслуживать как запросы на чтение, так и на запись.

Шардинг – разделение данных по узлам:

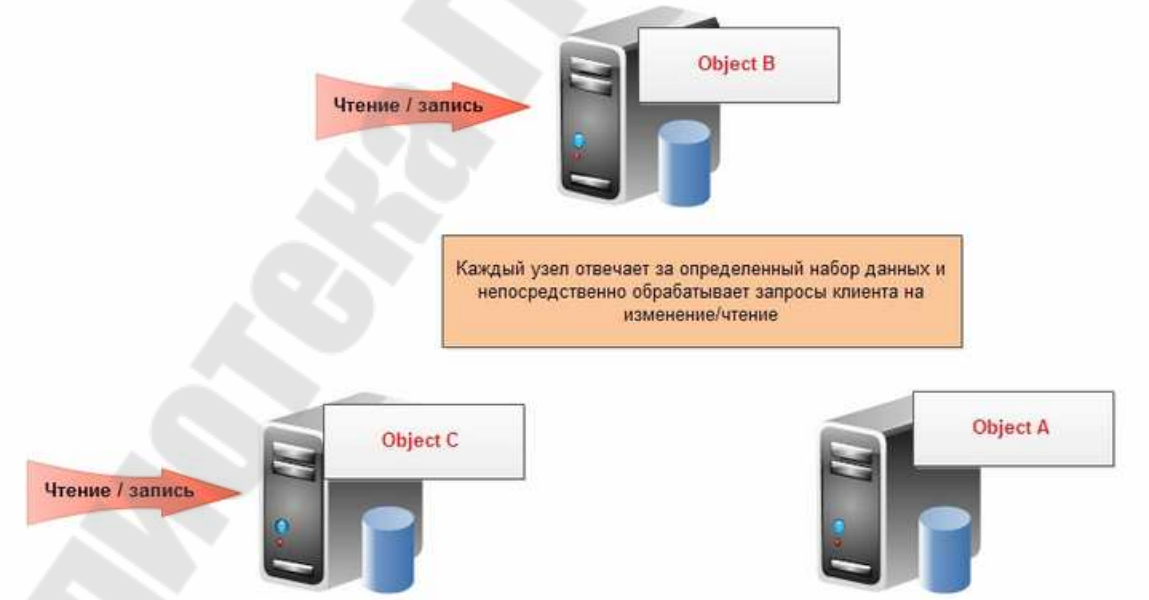

Рисунок 2.5. Шардинг

Шардинг часто использовался как “костыль” к реляционным базам данных в целях увеличения скорости и пропускной способности: пользовательское приложение партицировало данные по нескольким независимым базам данных и при запросе соответствующих данных пользователем обращалось к конкретной базе. В NoSQL базах данных шардинг, как и репликация, производятся автоматически самой базой и пользовательское приложение обособленно от этих сложных механизмов.

6. NoSQL базы в-основном оупенсорсные и созданы в 21 столетии.

Именно по второму признаку Садаладж и Фаулер не классифицировали объектные базы данных как NoSQL, так как они были созданы еще в 90-х и так и не снискали большой популярности.

Введение в SQL для Cosmos DB

Cosmos DB – мультимодельная база данных NoSql. В настоящее время она может обрабатывать три типа нереляционных данных:

1. Базы документов.
2. Графические базы данных.
3. Базы данных ключ-значение.

Только одна из этих моделей данных может быть запрошена с использованием SQL в базе данных Cosmos. Это база данных документов.

База данных документов – это нереляционная база данных, предназначенная для хранения документов, которые не имеют фиксированной структуры. Ведущие базы данных документов используют JavaScript Object Notation (JSON) в качестве формата для структурирования данных.

Пример данных

Структура JSON, которую будем использовать для отработки основ SQL-запросов с использованием Cosmos DB, полностью сглажена и выглядит следующим образом:

```
 {
"CountryName":"United Kingdom",
 "MakeName":"Ferrari",
 "ModelName":"Testarossa",
 "Cost":52000.0000,
```

```
  "RepairsCost":2175.0000,
  "PartsCost":1500.0000,
  "TransportInCost":750.0000,
  "Color":"Red",
  "SalePrice":65000.00,
  "LineItemDiscount":2700.00,
  "InvoiceNumber":"GBPGB001",
  "SaleDate":"2015-01-02T08:00:00",
  "CustomerName":"Magic Motors",
  "SalesDetailsID":1
}
```

Основные SQL-запросы

Начните с самого простого возможного примера. Запустив эмулятор Cosmos DB, введите и выполните следующий запрос:

```
SELECT * FROM C
```

Вы увидите все документы в текущей коллекции, возвращенные в качестве выходных данных, каждый из которых похож на образец документа, показанный выше.

Выполнение этого запроса возвращает только два доступных атрибута JSON. Здесь стоит отметить, что необходимо использовать имя коллекции или псевдоним при обращении к атрибутам. Обычно проще использовать короткие псевдонимы для коллекции, например:

```
SELECT s.InvoiceNumber, s.TotalSalePrice
FROM saleselements AS s
```

Ключевое слово AS используется, когда псевдонимы коллекций не является обязательными. Имена атрибутов чувствительны к регистру, хотя их неправильное написание не остановит выполнение запроса и предотвратит возвращение атрибута в вывод.

Конечно, вы можете добавить псевдонимы к атрибутам:

```
SELECT s.InvoiceNumber, s.SalePrice AS Price
FROM   saleselements AS s
```

Выполнение этого выдаст следующий JSON:

```
{
    "InvoiceNumber": "GBPGB011",
    "Price": 8500
}
```

Однако, когда создаются псевдонимы для атрибутов, необхоимо помнить, что:

– Ключевое слово AS является необязательным

– Псевдонимы чувствительны к регистру.

– Псевдонимы должны соответствовать соглашениям об именах JavaScript – вы не можете начинать псевдоним с цифры или другого не алфавитного символа, кроме знака доллара или подчеркивания

Конечно, спецификация JSON называет атрибуты «members», а данные для каждого члена – элементом, но использование атрибута word для описания имени фрагмента данных настолько распространено, что желательно его использовать.

Cosmos DB SQL будет возвращать строки, числа и даты в точности так, как они хранятся в документе JSON – как можно увидеть, если выполнить этот запрос:

```
SELECT   s.InvoiceNumber
    ,s.SaleDate
    ,s.SalePrice
FROM saleselements AS s
```

Результат будет следующим:

```
{
    "InvoiceNumber": "GBPGB011",
    "SaleDate": "2015-04-30T00:00:00",
    "SalePrice": 8500
}
```

Конечно, Cosmos DB SQL может выполнять основную арифметику:

```
SELECT   s.InvoiceNumber
    ,s.SalePrice - s.Cost AS GrossProfit
FROM saleselements AS s
```

Как вы можете видеть ниже:

```
{
    "InvoiceNumber": "GBPGB011",
    "GrossProfit": 1700
}
```

В качестве варианта темы вы всегда можете написать предложения SELECT следующим образом:

```
SELECT   s["InvoiceNumber"]
```

```
,s["SaleDate"]
FROM saleselements AS s
```

Действительно, можно смешивать и сопоставлять обозначения «атрибута точки псевдонима» и обозначения «квадратная скобка и двойная кавычка псевдонима» (в этом случае точки, используемые для связи псевдонима и имени атрибута) внутри одного и того же запроса.

Простые операторы WHERE

Время перейти к выбору данных. В конце концов, Cosmos DB может хранить терабайты документов JSON и позволяет масштабировать запросы для использования большей или меньшей вычислительной мощности, чтобы было возможно сбалансировать требования времени и стоимости запроса (в финансовом смысле) для удовлетворения каждого отдельного требования.

Основной текстовый фильтр

Предположим, что необходимо изолировать все счета, где был продан Bentley.

```
SELECT   s.InvoiceNumber ,s.SalePrice
FROM s
WHERE    s.MakeName = "Bentley"
```

Вы должны увидеть результат, подобный следующему:

```
{
    "InvoiceNumber": "GBPGB011",
    "SalePrice": 80500
}
```

Числовые фильтры

Фильтрация по числам в атрибутах JSON – это, вероятно, то, чего можно ожидать. Можно написать условие WHERE, подобное этому, чтобы указать точную цифру:

```
SELECT   s.InvoiceNumber ,s.SalePrice
FROM s
WHERE    s.LineitemNumber = 2
```

Или вот так, чтобы определить числовой диапазон:

```
SELECT   s.InvoiceNumber ,s.SalePrice
FROM s
```

```
WHERE    s.Cost BETWEEN 50000 AND 100000
```

Тогда результат будет следующим:

```
{
    "InvoiceNumber": "GBPGB011",
    "SalePrice": 8500
}
```

Также можно фильтровать поле и одновременно включать его в предложение SELECT, как показано ниже (хотя это будет указывать на достоверность фильтра в результатах):

```
SELECT   s.InvoiceNumber, s.Cost BETWEEN 50000 AND 100000
    AS InCostRange, s.SalePrice
FROM s
```

Этот запрос даст следующий результат:

```
{
    "InvoiceNumber": "GBPGB011",
    "InCostRange": false,
    "SalePrice": 8500
}
```

Алфавитные Диапазоны

Если когда-либо потребуется возвращать документы, основанные на алфавитном диапазоне в атрибуте, то одним из способов получения желаемого результата является использование такого вида SQL:

```
SELECT   s.CustomerName ,s.SalePrice
FROM saleselements AS s
WHERE    SUBSTRING(s.CustomerName, 0, 1) BETWEEN "A"
AND "M"
```

Фильтры даты и времени

Cosmos DB SQL допускает некоторые более продвинутые методы фильтрации в условии WHERE. Следует отметить, что хотя возможности значительно более ограничены, чем возможности, предлагаемые T-SQL, тем не менее можно включить (и исключить) нерелевантные документы JSON, отфильтровывая элементы даты и времени.

JSON не имеет типа даты как таковой. Вместо этого он использует строку. Следовательно, есть риск встретить даты в любом из множества форматов.

Формат в данных примера примерно соответствует формату ISO 8601, который выглядит следующим образом: ГГГГ-ММ-ДДЧЧ: ММ: СС. Это означает, что поиск определенной даты и времени означает представление даты и времени в виде строки ISO следующим образом:

```
SELECT s.SaleDate
FROM   s
WHERE  s.SaleDate = "2015-01-02T08:00:00"
```

Если необходимо установить верхнюю или нижнюю границу для даты или времени, то можно использовать стандартные операторы сравнения:> =, <=, <> или! =.

Использование диапазонов дат

Отсюда следует, что для определения диапазона дат не требуется ничего, кроме простого оператора AND в условии WHERE.

```
SELECT s.SaleDate
FROM   s
WHERE  s.SaleDate >= "2015-01-01T00:00:00"
  AND
  s.SaleDate <= "2015-01-31T23:59:59"
```

Это вернет данные из трех документов:

```
[
  {
    "SaleDate": "2015-01-02T08:00:00"
  },
  {
    "SaleDate": "2015-01-25T00:00:00"
  },
  {
    "SaleDate": "2015-01-02T10:33:00"
  }
]
```

Или, возможно, более элегантно, можно также использовать стандартную конструкцию BETWEEN … AND:

```
SELECT s.SaleDate
FROM   s
WHERE  s.SaleDate BETWEEN "2015-01-01T00:00:00"
  AND "2015-01-31T23:59:59"
```

Это возвращает данные из тех же трех документов:

```
[
  {
    "SaleDate": "2015-01-02T08:00:00"
  },
  {
    "SaleDate": "2015-01-25T00:00:00"
  },
  {
    "SaleDate": "2015-01-02T10:33:00"
  }
]
```

Применение фильтров времени

Фильтрация по времени, по сути, означает извлечение элементов часов, минут и, возможно, секунд или миллисекунд из строки даты. Итак, чтобы найти все автомобили, проданные в период с 8:45 до 12:00, необходимо написать:

```
SELECT s.SaleDate
FROM   s
WHERE  SUBSTRING(s.SaleDate, 11, 2) BETWEEN "08:45" AND
"12:00"
```

Это должно вернуть данные из двух документов в коллекции:

```
[
  {
    "SaleDate": "2015-02-03T10:00:00"
  },
  {
    "SaleDate": "2015-01-02T10:33:00"
  }
]
```

Отсутствие схемы в документах JSON означает, что дата может быть указана в совершенно другом строковом формате или даже в виде числа.

Ясно, что необходимые запросы должны быть адаптированы к используемому формату даты, и в этом случае теоретический запрос для совершенно другой коллекции (где даты хранились по-разному) может выглядеть следующим образом:

```
SELECT s.SaleDate
FROM   s
WHERE  s.SaleDate BETWEEN 20150101 AND 20150131
```

Опять же, свободная форма JSON в означает, что придется проявлять изобретательность и ловкость ума в написанных вами запросах.

Сортировка вывода

Cosmos DB SQL также содержит условие ORDER BY, поэтому можно писать следующие запросы:

```
SELECT   s.InvoiceNumber ,s.SalePrice AS Price
FROM saleselements AS s
ORDER BY s.InvoiceNumber
```

Вероятно, неудивительно, что увидите набор результатов, похожий на следующий:

```
{      "InvoiceNumber": "EURDE004",
       "Price": 11500
  },
  {      "InvoiceNumber": "EURFR005",
       "Price": 19950
  },
  {      "InvoiceNumber": "GBPGB001",
       "Price": 65000
  },
  {      "InvoiceNumber": "GBPGB002",
       "Price": 220000
}
```

Однако на данном этапе на горизонте появляется ряд ограничений. Поскольку, хотя возможно добавить ключевые слова ASC/DESC к условию ORDER BY, здесь применяются следующие ограничения:

– Ключевые слова ASCENDING и DESCENDING не поддерживаются

– Сортировка по нескольким атрибутам невозможна

– Нет возможности сортировать псевдоним для атрибута

– Нет возможности применять ORDER BY к чему-либо кроме имени поля

– Нет возможнсоти сортировать по вычисленному значению

2.7 Подходы к работе с неструктурированными данными

Ключевые различия между неструктурированными данными и структурированными данными.

Структурированные данные представляют собой высокоорганизованную, фактическую и точную информацию. Обычно он представлен в форме букв и цифр, которые хорошо вписываются в строки и столбцы таблиц. Структурированные данные обычно существуют в таблицах, подобных файлам Excel и электронным таблицам Google Docs.

Неструктурированные данные не имеют заранее определенной структуры и представлены во всем разнообразии форм. Примеры неструктурированных данных варьируются от изображений и текстовых файлов, таких как документы PDF, до видео и аудио файлов, и это лишь некоторые из них.

Структурированные данные часто называют количественными данными, что означает, что их объективный и заранее определенный характер позволяет нам легко подсчитывать, измерять и выражать данные в числах. Неструктурированные данные также называются качественными данными в том смысле, что они имеют субъективный и интерпретирующий характер. Эти данные можно разделить на категории в зависимости от их характеристик и свойств.

Существует широкий спектр форм, которые составляют неструктурированные данные, такие как электронная почта, текстовые файлы, сообщения в социальных сетях, видео, изображения, аудио, данные датчиков и так далее. Сообщение туристического агентства в Facebook: пример неструктурированных данных.

В качестве примера можно взять сообщения в социальных сетях туристического агентства или все публикации. Каждый пост содержит некоторые показатели, такие как репосты или хэштеги, которые можно количественно определить и структурировать. Однако сами посты относятся к категории неструктурированных данных. Т.е. для анализа сообщений и сбора полезной информации потребуется некоторое время, усилия, знания и специальные программные инструменты. Если агентство публикует новые туристические туры и хочет узнать реакцию аудитории (комментарии), им нужно будет изучить публикацию в ее собственном формате (просмотреть публикацию в

приложении социальных сетей или использовать передовые методы, такие как анализ настроений).

Структурированные данные обычно представлены в виде текста и чисел. Его форматы стандартизированы и удобочитаемы. Наиболее распространены CSV и XML. В модели данных формат данных был определен заранее.

В отличие от структурированных данных, неструктурированные форматы данных представлены в избытке различных форм и размеров. Неструктурированные данные не имеют заранее определенной модели данных и хранятся в своих *собственных форматах* (так называемых «исходных» форматах). Это могут быть аудио (WAV, MP3, OGG и т. Д.) Или видеофайлы (MP4, WMV и т. Д.), PDF-документы, изображения (JPEG, PNG и т. Д.), Электронные письма, сообщения в социальных сетях, данные датчиков и т. Д.

Структурированные данные менее гибкие, так как они основаны на строгой организации модели данных. Такие данные зависят от схемы. Схема базы данных обозначает конфигурацию столбцов (также называемых полями) и типы данных, которые должны храниться в этих столбцах. Такая зависимость является как преимуществом, так и недостатком. Хотя информацию здесь можно легко найти и обработать, все записи должны соответствовать очень строгим требованиям схемы.

С другой стороны, неструктурированные данные обеспечивают большую гибкость и масштабируемость. Отсутствие заранее определенной цели неструктурированных данных делает их очень гибкими, поскольку информация может храниться в различных форматах файлов. Однако эти данные субъективны и с ними труднее работать.

Если данные применяются для аналитической обработки и используются так называемые конвейеры данных, конечным пунктом путешествия структурированных данных будут специальные хранилища данных . Это компактные хранилища или репозитории с определенной структурой, которую сложно изменить. Даже незначительные изменения схемы могут привести к необходимости восстановления огромных объемов данных, что может потребовать затрат времени и ресурсов.

Чем больше объем данных, тем больше места требуется для хранения. Картинка с высоким разрешением весит намного больше, чем текстовый файл. Следовательно, неструктурированные данные

требуют больше места для хранения и обычно хранятся в озерах данных, репозиториях хранения, которые позволяют хранить практически неограниченные объемы данных в необработанных форматах. Помимо озер данных, неструктурированные данные хранятся в собственных приложениях.

Структурированные данные часто называют количественными данными. Это означает, что такие данные обычно содержат точные числа или текстовые элементы, которые можно подсчитать.

Неструктурированные данные, в свою очередь, часто классифицируются как качественные данные**,** содержащие субъективную информацию, которую невозможно обработать с помощью традиционных методов и инструментов программного анализа. Например, качественные данные могут поступать из опросов клиентов или отзывов в социальных сетях в текстовой форме. Для обработки и анализа качественных данных требуются более современные аналитические методы, такие как:

– накопление данных или исследование больших объемов данных, разделение их на более мелкие элементы и объединение переменных с одинаковыми значениями в одну группу, а также

– интеллектуальный анализ данных или процесс обнаружения определенных закономерностей, странностей и взаимодействий в больших наборах данных для предварительного выражения возможных результатов.

Говоря о базах данных для неструктурированных данных, наиболее подходящим вариантом для этого типа данных будут нереляционные базы данных, также известные как базы данных NoSQL.

NoSQL означает «не только SQL». Эти базы данных имеют различные модели данных и хранят данные в нетабличном виде. Наиболее распространенные типы баз данных NoSQL – это ключ-значение, документ, граф и широкий столбец. Такие базы данных могут обрабатывать огромные объемы данных и справляться с высокими пользовательскими нагрузками, поскольку они достаточно гибкие и масштабируемые. В мире NoSQL существуют скорее коллекции данных, чем таблицы. В этих сборниках есть так называемые документы. Хотя документы могут выглядеть как строки в таблицах, они не используют одну и ту же схему. В одной коллекции может быть несколько документов с разными полями. Кроме того, между элементами данных практически нет отношений. Идея здесь состоит в том, чтобы умень-

шить количество слияний отношений и вместо этого иметь сверхбыстрые и эффективные запросы. Хотя будут и дубликаты данных.

Одно из основных различий между структурированными и неструктурированными данными заключается в том, насколько легко их можно подвергнуть анализу. Структурированные данные в целом легко искать и обрабатывать, независимо от того, обрабатывает ли это человек или выполняет программные алгоритмы. Неструктурированные данные, напротив, гораздо сложнее искать и анализировать. После обнаружения такие данные должны быть внимательно обработаны, чтобы понять их ценность и применимость. Этот процесс сложен, поскольку неструктурированные данные не могут поместиться в фиксированные поля реляционных баз данных, пока они не будут собраны и обработаны.

С исторической точки зрения, поскольку структурированные данные существуют дольше, логично, что для них существует отличный выбор зрелых аналитических инструментов. В то же время те, кто работает с неструктурированными данными, могут столкнуться с меньшим выбором инструментов аналитики, поскольку большинство из них все еще разрабатываются. Использование традиционных инструментов интеллектуального анализа данных обычно разбивается о неорганизованную внутреннюю структуру этого типа данных.

Инструменты для неструктурированных данных

Поскольку неструктурированные данные бывают разных форм и размеров, для их правильного анализа и обработки требуются специально разработанные инструменты. Кроме того, необходимо найти квалифицированную команду по анализу данных. Мало того, что полезно понимать тему данных, но также важно выяснить отношения этих данных.

Рассмотрим несколько примеров инструментов и технологий для эффективного управления неструктурированными данными:

– *MongoDB*. Это система управления базами данных, ориентированная на документы, не требующая жесткой схемы или структуры таблиц. Он считается одним из классических примеров NoSQL. MongoDB использует документы, подобные JSON.

– *Amazon DynamoDB*. DynamoBD, предлагаемый Amazon в составе пакета AWS, представляет собой расширенную службу баз данных NoSQL для полного управления данными. Он поддерживает

структуры данных документов и значений типа “ключ-значение” и хорошо подходит для работы с неструктурированными данными.

– *Apache Hadoop*. Это эффективная среда с открытым исходным кодом, используемая для обработки больших объемов данных и их хранения на недорогих обычных серверах. Hadoop не только является мощным инструментом, но и гибким, поскольку не требует наличия схемы или структуры для хранимых данных. Он помогает структурировать неструктурированные данные и затем экспортировать эти данные в реляционные базы данных.

– *Microsoft Azure*. Представленный Microsoft, Azure представляет собой комплексную облачную службу для создания и управления приложениями и службами через центры обработки данных. Azure Cosmos DB – это быстрая и масштабируемая база данных NoSQL, которая помогает хранить и анализировать большие объемы неструктурированных данных.

В свое время анализ неструктурированных данных обычно выполнялся вручную и требовал много времени. В настоящее время существует довольно много продвинутых инструментов на основе ИИ, которые помогают сортировать неструктурированные данные, находить соответствующие элементы и сохранять результаты. Технологии и инструменты для неструктурированных данных включают алгоритмы обработки естественного языка и машинного обучения. Таким образом, можно адаптировать программные продукты к потребностям конкретных отраслей.

Благодаря тому, что реляционные базы данных существуют здесь дольше, они более знакомы пользователю. В отличие от инструментов структурированных данных, инструменты, предназначенные для неструктурированных данных, более сложны в работе. Следовательно, им требуется определенный уровень знаний в области науки о данных и машинного обучения для проведения глубокого анализа данных. Кроме того, специалисты, работающие с неструктурированными данными, должны хорошо разбираться в теме данных и их взаимосвязи. Учитывая вышесказанное, для обработки неструктурированных данных компании потребуется квалифицированная помощь специалистов по данным, инженеров и аналитиков**.**

Чаще всего отраслям необходимо использовать оба типа данных для повышения эффективности своих услуг.

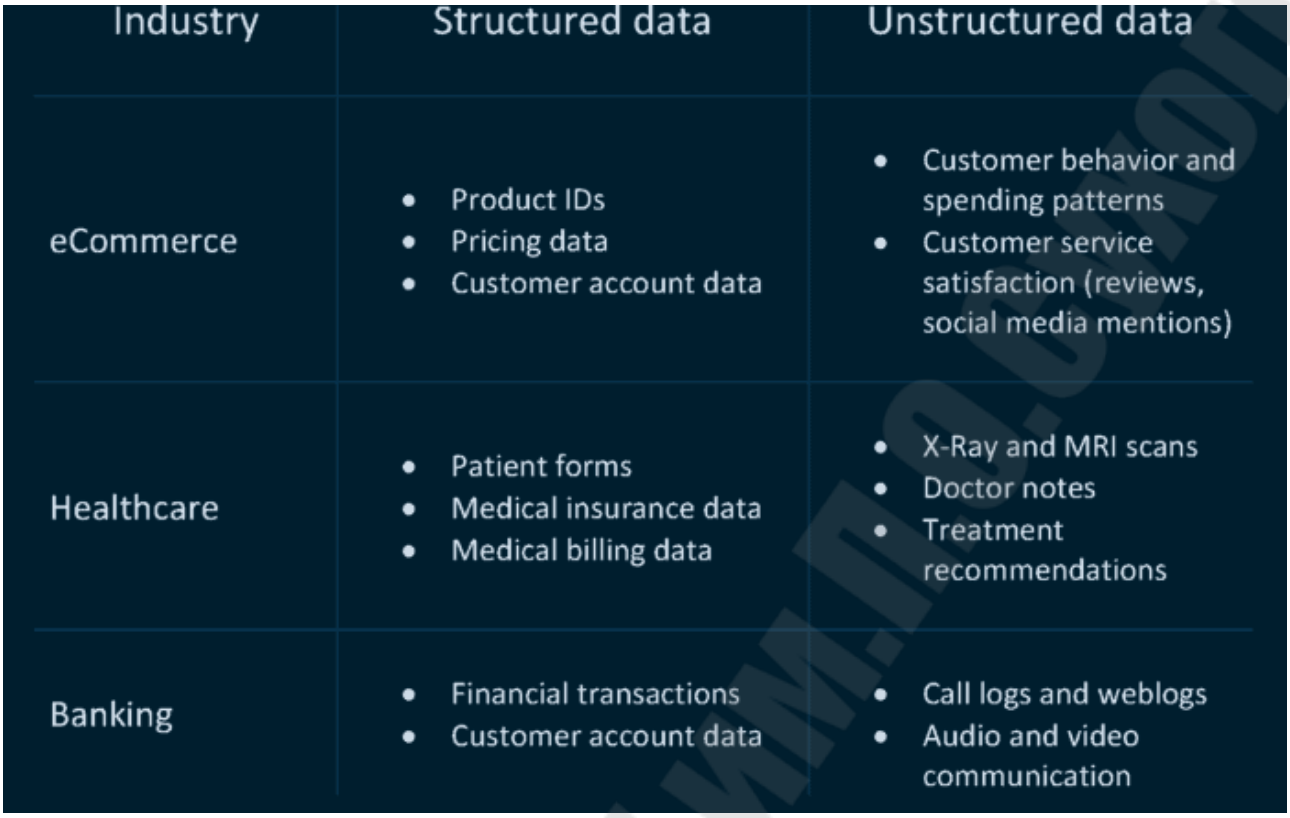

Рисунок 2.6.Использование структурированных
и неструктурированных данных в разных отраслях

Что такое полуструктурированные данные?

Как следует из названия, полуструктурированные данные частично структурированы, что означает, что они включают в себя определенные маркеры, которые могут разделять семантические элементы и реализовывать иерархии данных, но они все же отличаются от табличных моделей данных, представленных в реляционных базах данных. Такая структура называется самоописывающейся. Языки разметки, такие как XML, представляют собой формы полуструктурированных данных. JSON также является полуструктурированной моделью данных, которая используется базами данных нового поколения, такими как MongoDB и Couchbase. Существует множество других инструментов и решений для больших данных, которые используют эту категорию данных, потому что ее значительно проще обрабатывать, чем, скажем, неструктурированные данные.

Хотя полуструктурированные данные могут показаться золотой серединой, на самом деле это не так. В сегодняшней высококонкурентной среде предприятиям необходимо использовать все источники

данных для получения информации и правильно ее использовать, чтобы извлечь выгоду.

Подводя итоги, стоит сказать, что настоящей борьбы между неструктурированными и структурированными данными нет. Оба типа данных имеют большое значение для предприятий различных сфер и масштабов. Выбор источника данных может зависеть от структуры данных. Но чаще всего мы не выбираем один тип по сравнению с другим, а ищем возможности программного обеспечения для обработки всех данных.

В прошлом у компаний не было реального способа анализа неструктурированных данных, поэтому от них отказались, а основное внимание было уделено данным, которые можно было легко подсчитать. В настоящее время компании могут использовать искусственный интеллект, возможности машинного обучения и расширенную аналитику для выполнения за них сложного анализа неструктурированных данных. Например, такие корпорации, как Google, добились огромных успехов в технологии распознавания изображений, создав алгоритмы ИИ, которые могут автоматически определять, что или кто находится на фотографии.

По правде говоря, эти границы между структурированными и неструктурированными данными немного размыты, потому что в наши дни большинство наборов данных частично структурированы. Даже если возьмем неструктурированные данные, такие как фотография, они все равно будут содержать компоненты структурированных данных, такие как размер изображения, разрешение, дата создания изображения и т. д. Эта информация может быть организована в табличном формате реляционных баз данных.

Анализ неструктурированных данных и оптимизация их хранения

Тема анализа неструктурированных данных сама по себе не нова. Однако в последнее время в эпоху «больших данных» этот вопрос встает перед организациями гораздо острее. Многократный рост объемов хранимых данных в последние годы, его постоянно увеличивающиеся темпы и нарастающее разнообразие хранимой и обрабатываемой информации существенно усложняют задачу управления корпоративными данными. С одной стороны, проблема имеет инфраструктурный характер. Так, по данным IDC, до 60% корпоративных хранилищ занимает информация, не приносящая организации ника-

кой пользы (многочисленные копии одного и того же, разбросанные по разным участкам инфраструктуры хранения данных; информация, к которой никто не обращался несколько нет и уже вряд ли когда-нибудь обратится; прочий «корпоративный мусор»).

С другой стороны, неэффективное управление информацией ведет к увеличению рисков для бизнеса: хранение персональных данных и прочей конфиденциальной информации на общедоступных информационных ресурсах, появление подозрительных пользовательских зашифрованных архивов, нарушения политик доступа к важной информации и т. д.

В этих обстоятельствах умение качественно анализировать корпоративную информацию и оперативно реагировать на любые несоответствия ее хранения политикам и требованиям бизнеса является ключевым показателем зрелости информационной стратегии организации.

Теме аналитики файловых данных посвящен отдельный документ Gartner, вышедший в сентябре 2014 г. под названием «Market Guide for File Analysis Software». В данном документе приводятся следующие типовые сценарии использования аналитического ПО:

1. Оптимизация хранения. Наиболее типичный сценарий. Целью внедрения файловой аналитики является снижение объема хранимых данных, и, тем самым, повышение эффективности их хранения.

2. Выявление ненужных данных и избавление от них при миграции ИТ-инфраструктуры. Часто инициируются проектами по миграции данных в «облако». Проводится сканирование контента и по его результатам имеющие важность и ценность для бизнеса данные «переезжают» в «облако», а остальные удаляются.

3. Классификация. Целью таких проектов по анализу является группировка объектов по различным критериям для назначения на них общих политик, понимания ценности и потенциального риска, которые несет хранимая информация.

4. Соблюдение нормативов и требований (compliance). Специалисты соответствующих подразделений могут разработать и внедрить политики доступа к важным данным и за счет встроенной в аналитическое ПО классификации эффективно контролировать их соблюдение.

5. Управление уровнями доступа. За счет получения информации об уровне и типе доступа пользователей к файлам и директориям возможно осуществлять информационный менеджмент с целью за-

щиты персональных данных и иной конфиденциальной информации от несанкционированного доступа.

6. Автоматизация проведения расследований. Аналитическое ПО позволяет быстро находить объекты, имеющие отношение к проводимым в компании расследованиям, и автоматизированно и безопасно копировать или перемещать их в специальные хранилища.

Принимая во внимание данный тренд, компания Hewlett-Packard выпустила на рынок два программных решения, предназначенных для расширенного анализа неструктурированной информации: HP Storage Optimizer и HP Control Point. Первое решение в основном предназначено для специалистов, отвечающих за хранение данных. Второе решение подойдет не только специалистам ИТ-подразделений, но будет также интересно сотрудникам отделов информационной безопасности, Compliance-служб, а также руководству, определяющим стратегию хранения и использования информации в организации.

HP Storage Optimizer: анализ данных с целью оптимизации их хранения

HP Storage Optimizer (рисунок 2.7) объединяет в себе возможности по анализу метаданных объектов в репозиториях неструктурированной информации и назначению политик их иерархического хранения.

Источники анализируемой информации в терминологии HP Storage Optimizer называются репозиториями. В качестве репозиториев поддерживаются различные файловые системы, а также MS Exchange, MS SharePoint, Hadoop, Lotus Notes, Documentum и многие другие. Есть также возможность заказать разработку коннектора к репозиторию, который в настоящее время не поддерживается продуктом.

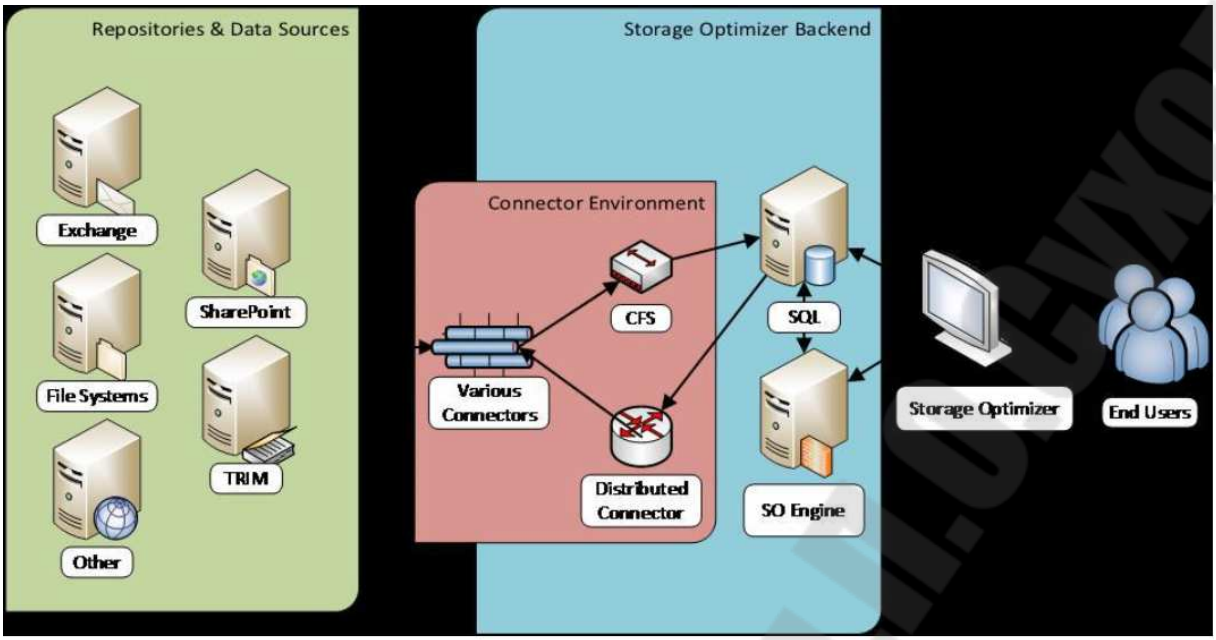

Рисунок 2.7. Архитектура HP Storage Optimizer

HP Storage Optimizer использует собственные соответствующие коннекторы для обращения к анализируемым репозиториям. Информация с коннекторов поступает в компонент под названием Connector Framework Server (обозначенный как «CFS» на картинке), который, в свою очередь, обогащает ее дополнительными метаданными и направляет получившиеся данные на индексирование. Для повышения отказоустойчивости и балансировки нагрузки при взаимодействии приложения с коннекторами используется компонент Distributed Connector.

Метаданные индексируются «движком» HP Storage Optimizer Engine («SO Engine» на первой картинке) и помещаются в БД MS SQL. Для доступа к результатам анализа и назначения политик управления используется веб-приложение HP Storage Optimizer.

Для наглядного отображения информации, потенциально подлежащей оптимизации, в HP Storage Optimizer используются круговые диаграммы (ниже) (рисунок 2.8), показывающие дубликаты данных, редковостребованные и «ненужные» данные (ROT analysis: Redundant, Obsolete, Trivial). Критерии «редковостребованности» и «ненужности» можно гибко настроить, в том числе индивидуально для каждого репозитория. Кроме круговых диаграмм, доступны графики, иллюстрирующие разбивку данных по типам, времени и частоте добавления и др. Все элементы визуализации интерактивны,

т. е. позволяют переходить в какую-либо категорию диаграммы (или столбец) и получать доступ к соответствующим данным.

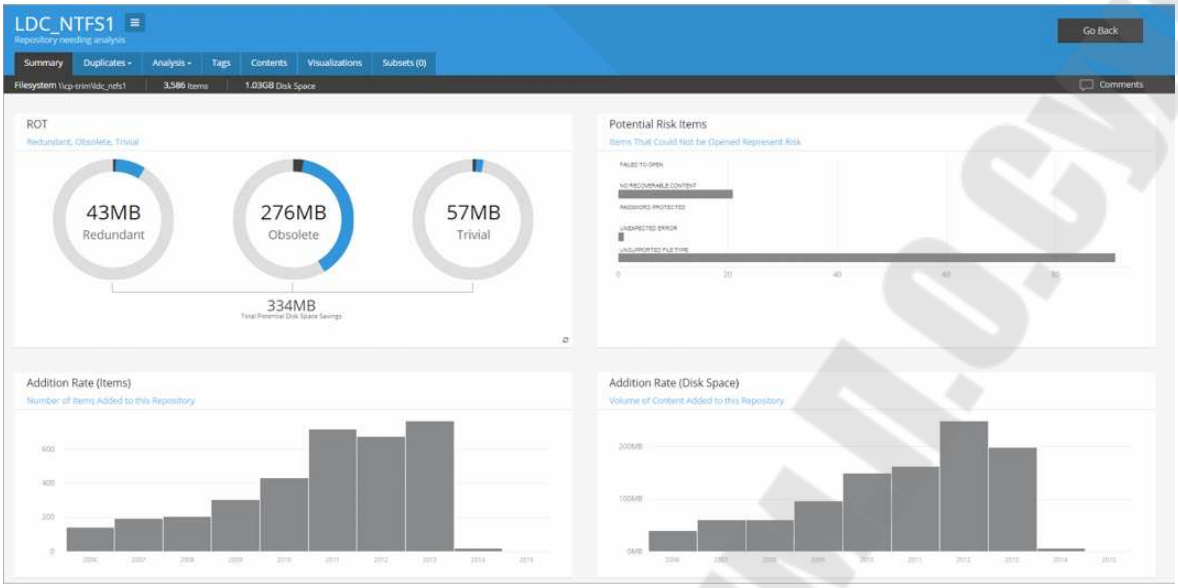

Рисунок 2.8. Графический анализ данных в HP Storage Optimizer

Перечень метаданных (рисунок 2.9), по которым может быть проведен анализ, необычайно широк и дает возможность осуществлять высокоточные тематические выборки.

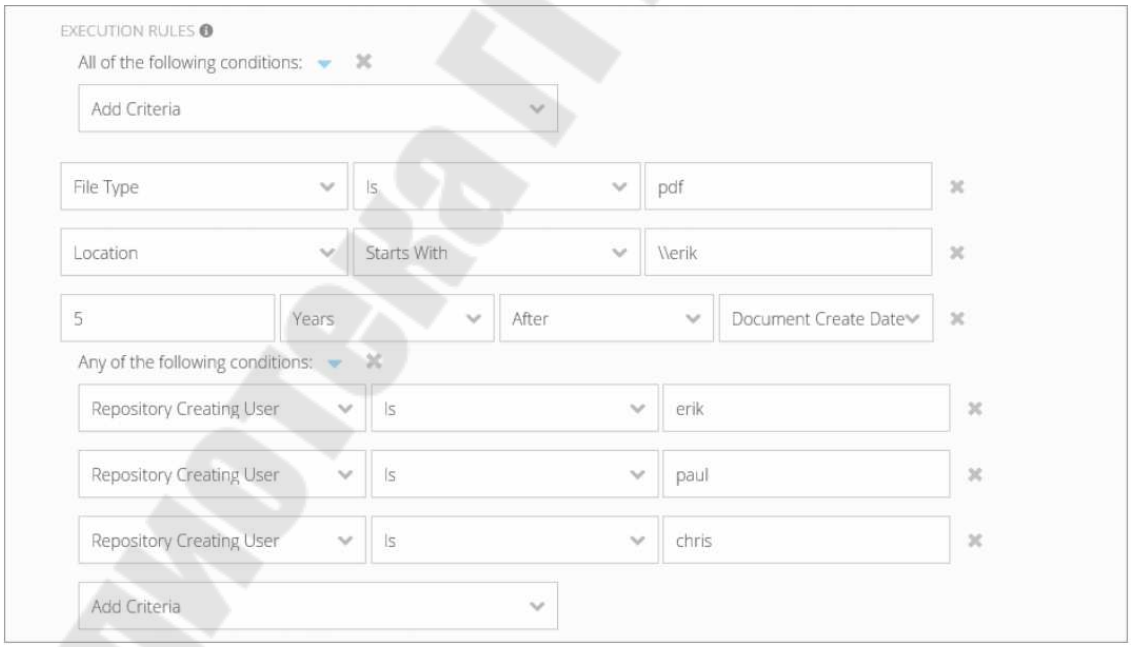

Рисунок 2.9. Пример работы с метаданными в HP Storage Optimizer

Хотелось бы заметить, что в состав продуктов HP Storage Optimizer и HP Control Point входит «движок» индексирования и визуализации, позволяющий просматривать более 400 различных форматов данных без установки на сервер соответствующих приложений для предпросмотра. Это значительно упрощает и ускоряет процесс анализа большого количества разноплановой информации.

После того как анализ данных проведен, администратору системы предоставляется возможность назначить политики удаления или перемещения данных. Политики на те или иные выборки данных возможно назначать как вручную, так и автоматически. Мощная ролевая модель управления, реализованная в HP Storage Optimizer и в HP Control Point, дает возможность выдавать полномочия по работе с репозиториями, анализу данных в них, а также по назначению политик, максимально гибко.

HP Control Point: комплексный анализ для снижения бизнес-рисков, связанных с хранением данных

HP Control Point, по сути, представляет собой расширенную версию HP Storage Optimizer и предоставляет инструментарий не только для решения задач по оптимизации хранения, но и для внедрения политик хранения и управления жизненным циклом корпоративной информации.

Продукт позволяет проводить анализ информации не только по метаданным, но и по ее содержимому. Кроме того, в нем реализованы дополнительные механизмы анализа данных и назначения политик по работе с ними.

В отличие от HP Storage Optimizer, в HP Control Point (рисунок 2.10) широко используются возможности индексирования и смысловой категоризации информации «движка» HP IDOL (Intelligent Data Operating Layer): визуализация, категоризация, тэгирование и др. В его основе лежит возможность определять «смысл» набора анализируемой информации независимо от ее формата, языка и т. д.

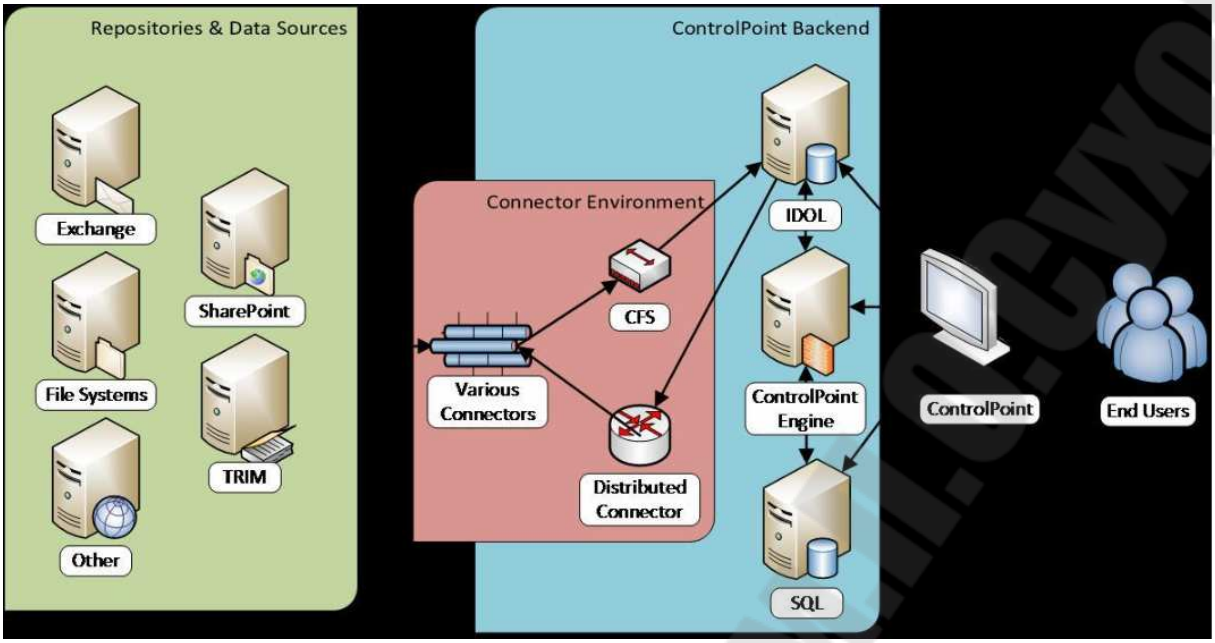

Рисунок 2.10. Архитектура HP Control Point

В частности, в HP Control Point дополнительно доступны два типа визуализации информации: кластерная карта (рисунок 2.11) и спектрограф (рисунок 2.12). Кластерная карта представляет собой двухмерное изображение информационных «кластеров». Один кластер объединяет в себе информацию, имеющую схожий смысл. Таким образом, глядя на кластерную карту, можно быстро получить понимание основных смысловых групп этой информации. Кластерные карты интерактивны, т. е. позволяют с помощью кликов на те или иные кластеры получать доступ к информации, содержащейся в них.

Спектрограф представляет собой набор информационных кластеров, снятых в различные моменты времени и дает возможность графически отследить, как менялся смысл информации в анализируемых репозиториях с течением времени.

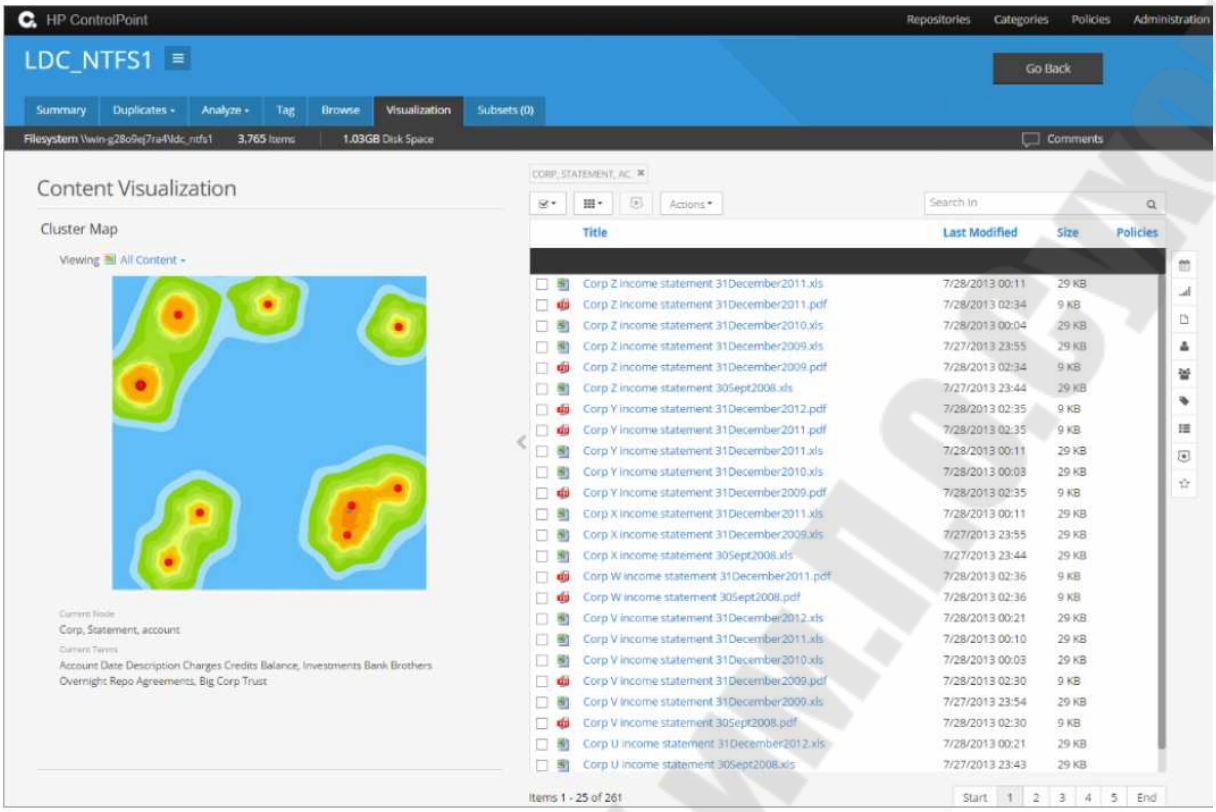

Рисунок 2.11. Внешний вид кластерной карты в HP Control Point

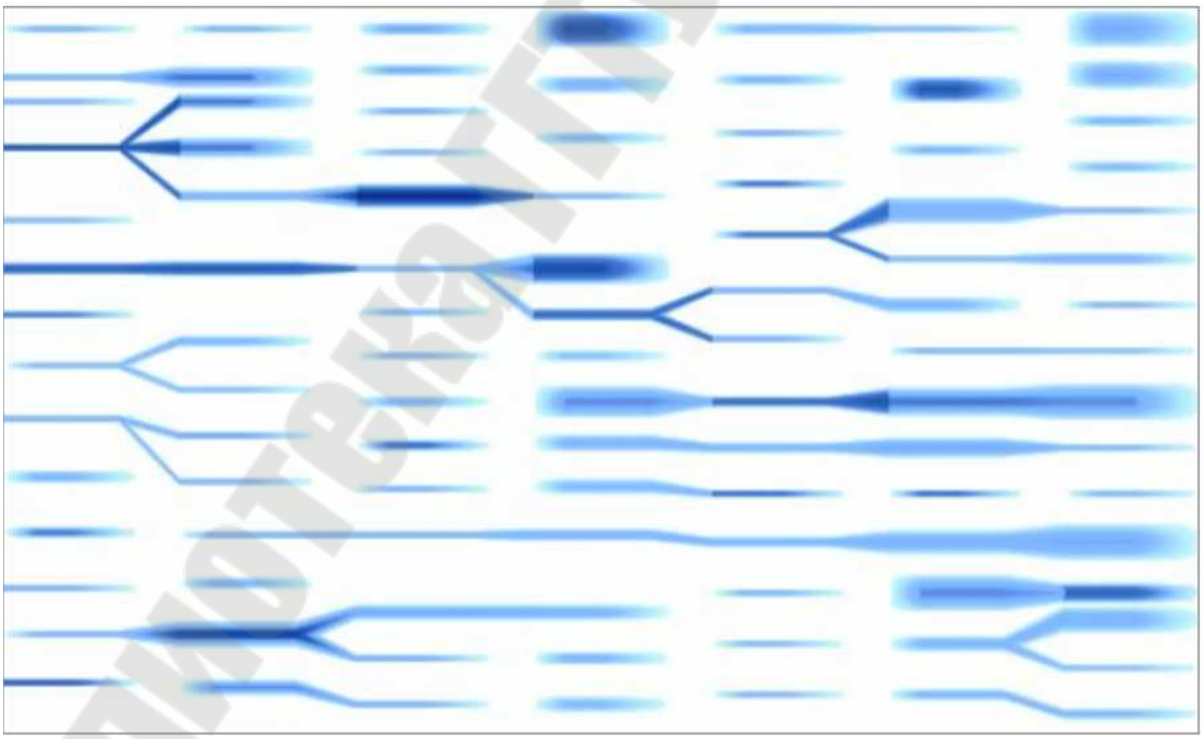

Рисунок 2.12. Внешний вид спектрограммы в HP Control Point

Помимо расширенных возможностей визуализации информации, в HP Control Point доступна возможность категоризации анализируемой информации. Изначально информация категоризируется автоматически – средствами HP IDOL, выдавая пользователю системы массив данных, разбитый на смысловые части. Получив первичное разбиение, аналитик далее может сделать более выверенную категоризацию. Например, использовать какой-либо набор файлов, заведомо для аналитика релевантных той или иной категории, для «тренировки» категории на этот набор файлов, чтобы впоследствии получать более точные результаты категоризации. Для еще более тонкой настройки можно использовать индивидуальные весовые коэффициенты файлов и даже фраз и отдельных слов внутри файлов, отражающие степень соответствия тех или иных единиц информации «тренируемой» категории. Такая детализация может использоваться, например, для создания подробных правил отнесения анализируемой информации к разряду конфиденциальной.

Что касается политик работы с анализируемой информацией, то в HP Control Point кроме копирования, переноса и удаления доступны также следующие опции:

– «Заморозка» объектов. Позволяет заблокировать доступ к отдельным объектам, не допуская их несанкционированное изменение или удаление.

– Создание рабочего процесса (workflow). Например, информирование или запрос утверждения уполномоченного сотрудника или владельца анализируемых объектов перед их переносом или удалением.

– Безопасный перенос в систему управления корпоративными записями HP Records Manager. При этом переносимые данные сопровождаются метаданными, которые будут использованы для дальнейшего управления документами в системе HP Records Manager с необходимыми настройками доступа, уровнями секретности и т.п.

Как можно заметить, спектр применения HP Storage Optimizer и HP Control Point для решения задач анализа и управления корпоративными данными весьма широк. Кроме того, возможности анализа документов на разных языках (включая русский), а также масштабируемая архитектура компонентов обоих продуктов позволяет эффективно решать задачи по анализу всего объема неструктурированных данных в организациях любого масштаба и сложности.

ЛИТЕРАТУРА

1. Allen, C. Oracle Database 10g PL/SQL 101 = Аллен К. Устная база данных 10 g PL| SQL 101 / C. Allen . – New-Delhi : Tata McGraw-Hill Publishing Company Limited, 2004. – 395 p.
2. Deshpande, P. S. SQL & PL/SQL for Oracle 10 g = Дифанде П. С. SQL & PL/SQL for Oracle 10g / P. S. Deshpande. – New Delhi : Dreamlech Press, 2010. – 980 p.
3. Nielsen, P. Microsoft SQL Server 2008 Bible = Нэльсон П. Microsoft SQL Server 2008 Библия / P. Nielsen, M. White, U. Parui. – New Delhi : Wiley India Pvt Ltd, 2009. – 1642 p.
4. Базы данных : учеб. для высш. учеб. заведений / А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев ; под ред. А. Д. Хомоненко. – 6-е изд. – М. : Бином-Пресс ; Санкт-Петербург : КОРОНА-Век, 2007. – 736 с.
5. Балджы, А. С. Математика на Python : учеб.-метод.пособие / А. С. Балджы, М. Б. Хрипунова, И. А. Александрова ; Финн. ун-т при Правительстве Рос. Федерации. – М. : Прометей, 2018. – Ч. 1. Элементы линейной алгебры и аналитической геометрии. – 76 с. : табл. – Режим доступа: https://biblioclub.ru/index.php?page=book&id=494849
6. Бейли, Л. Изучаем SQL = Head First SQL : пер. с англ. / Л. Бейли. – СПб : Питер, 2012. – 582 с.
7. Макленнен, Дж. Microsoft SQL Server 2008. Data Mining - интеллектуальный анализ данных : перевод с английского / Д. Макленнен, Ч. Танг, Б. Криват. – СПб : БХВ-Петербург, 2009. – 700 с.
8. Маркин, А. В. Построение запросов и программирование на SQL : учеб. пособие / А. В. Маркин. – 3-е изд., перераб. и доп. – М. : Диалог-МИФИ, 2014. – 384 с. : ил. – Режим доступа: https://biblioclub.ru/index.php?page=book&id=89077.
9. Полубояров, В.В. Использование MS SQL Server Analysis Services 2008 для построения хранилищ данных: курс / В. В. Полубояров. – М. : ИНТУИТ, 2010. – 586 с. : ил. – Режим доступа: https://biblioclub.ru/index.php?page=book&id=234554 .
10. Стасышин, В. М. Практикум по языку SQL : учеб. пособие / В. М. Стасышин, Т. Л. Стасышина ; Новосиб. гос. техн. ун-т. – Новосибирск : НГТУ, 2016. – 60 с. : ил., табл. – Режим доступа: https://biblioclub.ru/index.php?page=book&id=576764.

11. Сузи, Р. А. Язык программирования Python : учеб. пособие / Р. А. Сузи. – 2-е изд., испр. – М. : ИНТУИТ : Бином. Лаб. зн., 2007. – 327 с. – (Основы информационных технологий). – Режим доступа: https://biblioclub.ru/index.php?page=book&id=233288.

12. Хендерсон К. Профессиональное руководство по SQL Server : хранимые процедуры, HTML и XML / К. Хендерсон. – СПб : Питер, 2005. – 619 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ В АНАЛИЗ БОЛЬШИХ ДАННЫХ

Учебно-методическое пособие
для студентов специальностей
1-40 05 01 «Информационные системы и технологии
(по направлениям)» и 1-40 80 04 «Информатика
и технологии программирования»
дневной и заочной форм обучения

Составитель **Комракова** Евгения Владимировна

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 03.04.23.
Рег. № 13Е.
http://www.gstu.by