**Лабораторная работа №8**

**Оптимизация sql-запросов в СУБД с использованием планировщика**

**Цель работы:** получение навыков повышения производительности работы СУБД с помощью оптимизации sql-запросов с использованием планировщика.

**Основные теоретические сведения**

Метод доступа характеризует тот способ, который используется для просмотра таблиц и извлечения только тех строк, которые соответствуют критерию отбора. Существуют различные методы доступа: последовательный просмотр (*sequential scan*), при котором индекс не используется, и группа методов, основанных на использовании индекса. К ней относятся: просмотр по индексу (*index scan*), просмотр исключительно на основе индекса (*index only scan*) и просмотр на основе битовой карты (*bitmap scan*).

Другим важным понятием является способ соединения наборов строк (*join*). Набор строк может быть получен из таблицы с помощью одного их методов доступа, описанных выше. Набор строк может быть получен не только из одной таблицы, а может быть результатом соединения других наборов. Важно различать способ соединения таблиц (JOIN) и способ соединения наборов строк. Первое понятие относится к языку SQL и является высокоуровневым, логическим, оно не касается вопросов реализации. А второе относится именно к реализации, это – механизм непосредственного выполнения соединения наборов строк. Существует три способа соединения: вложенный цикл (*nested loop*), хеширование (*hash join*) и слияние (*merge join*). Они имеют свои особенности, которые СУБД учитывает при выполнении конкретных запросов.

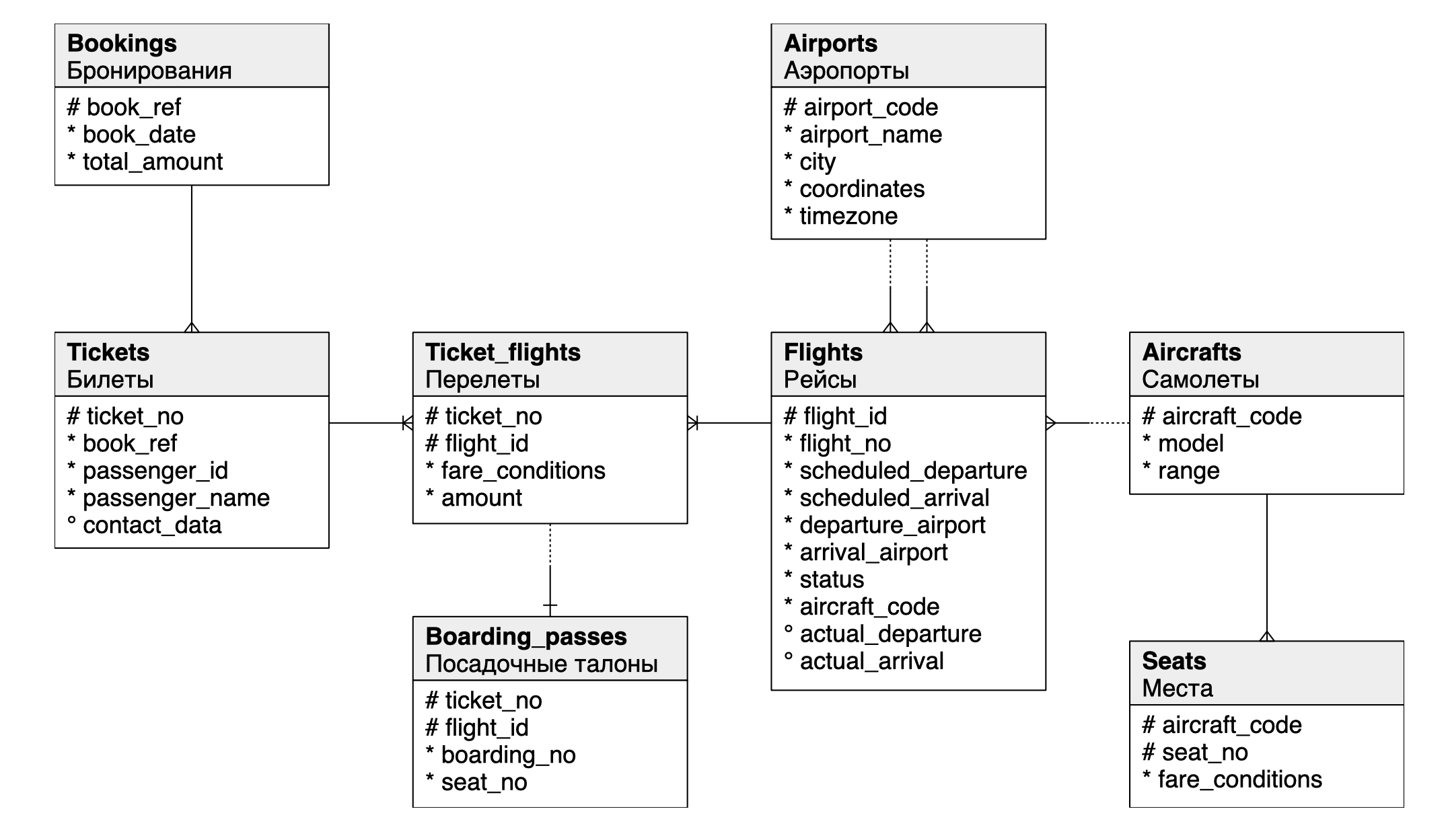
Структура плана запроса представляет собой дерево, состоящее из так называемых узлов плана (*plan nodes*). Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше. Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана. Например, для соединения наборов строк будут использоваться способы, которые рассмотрены выше. Для каждого узла дерева плана команда *EXPLAIN* выводит по одной строке, при этом выводятся также оценки стоимости выполнения операций на каждом узле, которые делает планировщик. В случае необходимости для конкретных узлов могут выводиться дополнительные строки.

**Задание к работе**

1. Сгенерировать данные для таблиц, используемых в sql-запросах лабораторной работы №4 (1000 записей).
2. Построить планы исполнения запросов, созданных в процессе выполнения лабораторной работы №4.
3. Повлиять на производительность с использованием методов управления планировщиком. Предложить способы ускорения выполнения запросов. Продемонстрировать планы исполнения запросов.
4. Провести сравнительный анализ выполнения запросов при 100 / 1000 записей в таблицах.

**Пример**

Пусть предметная область описывается следующей моделью данных:



В качестве примера, рассмотрим следующую задачу. Предположим, что необходимо определить степень загруженности кассиров нашей авиакомпании в декабре 2021 г. Для этого, в частности, требуется выявить распределение числа операций бронирования по числу билетов, оформленных в рамках этих операций. Другими словами, это означает, что нужно подсчитать число операций бронирования, в которых был оформлен только один билет, число операций, в которых было оформлено два билета и т. д.

Эту задачу можно переформулировать так: для каждой строки, отобранной из таблицы «Бронирования» (*bookings*), нужно подсчитать соответствующие строки в таблице «Билеты» (*tickets*). Речь идет о строках, в которых значение поля *book\_ref* такое же, что и в текущей строке таблицы *bookings*. Буквальное следование такой формулировке задачи приводит к получению запроса с коррелированным подзапросом в предложении *SELECT*. После этого требуется сгруппировать полученный набор строк по значениям числа оформленных билетов.Получаем следующий запрос:

EXPLAIN

SELECT num\_tickets, count( \* ) AS num\_bookings

FROM

( SELECT b.book\_ref,

( SELECT count( \* ) FROM tickets t

WHERE t.book\_ref = b.book\_ref

)

FROM bookings b

WHERE date\_trunc( 'mon', book\_date ) = '2021-12-01'

) AS count\_tickets( book\_ref, num\_tickets )

GROUP by num\_tickets

ORDER BY num\_tickets DESC;

QUERY PLAN

-----------------------------------------------------------------------

GroupAggregate (cost=14000017.12..27994373.35 rows=1314 width=16)

Group Key: ((SubPlan 1))

-> Sort (cost=14000017.12..14000020.40 rows=1314 width=8)

Sort Key: ((SubPlan 1)) DESC

-> Seq Scan on bookings b

(cost=0.00..13999949.05 rows=1314 width=8)

Filter: (date\_trunc('mon'::text, book\_date) = '2016-09-01

00:00:00+08'::timestamp with time zone)

SubPlan 1

-> Aggregate (cost=10650.17..10650.18 rows=1 width=8)

-> Seq Scan on tickets t

(cost=0.00..10650.16 rows=2 width=0)

Filter: (book\_ref = b.book\_ref)

(10 строк)В плане получены очень большие оценки общей стоимости выполнения запроса:

cost=14000017.12..27994373.35

Планировщик предполагает, что из таблицы tickets в подзапросе будет извлекаться всего по две строки, и эту операцию нужно будет проделать 1 314 раз: столько строк предположительно будет выбрано из таблицы bookings. Как видно из плана, для просмотра строк в таблице tickets используется ее последовательное сканирование. В результате оценка стоимости этого узла плана получается высокой: cost=0.00..10650.16

Создадим индекс для таблицы tickets по столбцу book\_ref, по которому происходит поиск в ней.

CREATE INDEX tickets\_book\_ref\_key

ON tickets ( book\_ref );

Повторим запрос, добавив параметр ANALYZE в команду EXPLAIN. Новый план, в котором отражены и фактические результаты, будет таким:

QUERY PLAN

-------------------------------------------------------------------------

GroupAggregate (cost=22072.70..38484.52 rows=1314 width=16)

(actual time=3656.554..3787.562 rows=5 loops=1)

Group Key: ((SubPlan 1))

-> Sort (cost=22072.70..22075.99 rows=1314 width=8)

(actual time=3656.533..3726.969 rows=165534 loops=1)

Sort Key: ((SubPlan 1)) DESC

Sort Method: external merge Disk: 2912kB

-> Seq Scan on bookings b (cost=0.00..22004.64 rows=1314 width=8)

(actual time=0.219..3332.162 rows=165534 loops=1)

Filter: (date\_trunc('mon'::text, book\_date) = '2016-09-01 00:00:00+08'::timestamp with time zone)

Rows Removed by Filter: 97254

SubPlan 1

-> Aggregate (cost=12.46..12.47 rows=1 width=8)

(actual time=0.016..0.016 rows=1 loops=165534)

-> Index Only Scan using tickets\_book\_ref\_key on tickets t

(cost=0.42..12.46 rows=2 width=0)

(actual time=0.013..0.014 rows=1 loops=165534)

Index Cond: (book\_ref = b.book\_ref)

Heap Fetches: 230699

Planning time: 0.290 ms

Execution time: 3788.690 ms

(15 строк)

Вывод: в процессе выполнения лабораторной работы были получены навыки оптимизации запросов к БД путем модификации текста запросов, а также добавления индексов.

Теперь планировщик использует индекс для поиска в таблице tickets. Причем поиск исключительно по индексу (Index Only Scan), поскольку нас интересует только число строк — count( \* ), а не их содержание. При этом запрос стал выполняться – на порядок быстрее.Кроме создания индекса есть и другой способ: замена коррелированного подзапроса соединением таблиц.EXPLAIN ANALYZE

SELECT num\_tickets, count( \* ) AS num\_bookings

FROM

( SELECT b.book\_ref, count( \* )

FROM bookings b, tickets t

WHERE date\_trunc( 'mon', b.book\_date ) = '2016-09-01'

AND t.book\_ref = b.book\_ref

GROUP BY b.book\_ref

) AS count\_tickets( book\_ref, num\_tickets )

GROUP by num\_tickets

ORDER BY num\_tickets DESC;

QUERY PLAN

------------------------------------------------------------------------

GroupAggregate (cost=16966.67..16978.53 rows=200 width=16)

(actual time=4092.258..4219.477 rows=5 loops=1)

Group Key: count\_tickets.num\_tickets

-> Sort (cost=16966.67..16969.96 rows=1314 width=8)

(actual time=4092.236..4161.294 rows=165534 loops=1)

Sort Key: count\_tickets.num\_tickets DESC

Sort Method: external merge Disk: 2912kB

-> Subquery Scan on count\_tickets

(cost=16858.57..16898.61 rows=1314 width=8)

(actual time=3176.113..3862.133 rows=165534 loops=1)

-> GroupAggregate (cost=16858.57..16885.47 rows=1314 width=15)

(actual time=3176.111..3765.157 rows=165534 loops=1)

Group Key: b.book\_ref

-> Sort (cost=16858.57..16863.16 rows=1834 width=7)

(actual time=3176.098..3552.334 rows=230699 loops=1)

Sort Key: b.book\_ref

Sort Method: external merge Disk: 3824kB

-> Hash Join (cost=5632.24..16759.16 rows=1834 width=7)

(actual time=498.701..1091.509 rows=230699 loops=1)

Hash Cond: (t.book\_ref = b.book\_ref)

-> Seq Scan on tickets t

(cost=0.00..9733.33 rows=366733 width=7)

(actual time=0.047..170.792 rows=366733 loops=1)

-> Hash (cost=5615.82..5615.82 rows=1314 width=7)

(actual time=498.624..498.624 rows=165534 loops=1)

Buckets: 262144 (originally 2048)

Batches: 2 (originally 1)

Memory Usage: 3457kB

-> Seq Scan on bookings b

(cost=0.00..5615.82 rows=1314 width=7)

(actual time=0.019..267.728 rows=165534 loops=1)

Filter: (date\_trunc('mon'::text, book\_date) =

'2016-09-01 00:00:00+08'::timestamp with time zone)

Rows Removed by Filter: 97254

Planning time: 2.183 ms

Execution time: 4221.133 ms

(21 строка)

В данном плане используется соединение хешированием (*Hash Join*). При этом индекс по таблице *tickets* игнорируется: таблица просматривается последовательно (*Seq Scan on tickets t*). Время выполнения модифицированного запроса оказывается несколько большим, чем в предыдущем случае, когда в запросе присутствовал коррелированный подзапрос. Таким образом, можно заключить, что для ускорения работы оригинального запроса можно было либо создать индекс, либо модифицировать сам запрос, даже не создавая индекса.