Федеральное государственное автономное образовательное учреждение
высшего образования
"Национальный исследовательский университет
"Высшая школа экономики"

Московский институт электроники и математики им. А.Н Тихонова

Департамент компьютерной инженерии

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ИНДЕКСОВ В РЕЛЯЦИОННЫХ БАЗАХ ДАННЫХ

Методические указания к лабораторной работе № 5 по курсу "Базы данных"

Москва

2024

Составитель: доцент, канд. техн. наук И.П. Карпова

УДК 681.3

Создание и использование индексов в реляционных базах данных: Методические указания к лабораторной работе № 5 по курсу "Базы данных" / Московский институт электроники и математики НИУ ВШЭ; Сост.: И.П. Карпова. – М., 2024. – 33 с.

Лабораторная работа посвящена изучению правил создания и использования индексов для ускорения выполнения запросов на языке SQL. Работа может быть выполнена под управлениям СУБД Oracle или PostgreSQL.

Для студентов II-IV курсов технических факультетов, изучающих автоматизированные информационные системы и системы баз данных.

Ил.: 25. Библиогр.: 5 назв.

Содержание

ЦЕЛЬ ВЫПОЛНЕНИЯ РАБОТЫ

Цель выполнения лабораторной работы – изучение основ создания и использования индексов для ускорения выполнения запросов к реляционным базам данных (БД) и получение практических навыков работы с индексами.

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Общие положения

Основным объектом реляционной базы данных является таблица. Строки таблицы называются *записями* или кортежами, столбцы таблицы – *атрибутами* (или полями записи). Одним из важных объектов реляционных (и не только) СУБД являются индексы. Индексирование – это способ доступа к данным в реляционной таблице с помощью специальной структуры – индекса.

Индекс – это структура, которая хранит в отсортированном виде значения атрибута записи (или группы атрибутов, например, ФИО) и местоположения этой записи (Рис. 1). По сути, индекс похож на предметный указатель в книге. Такой указатель содержит термины, используемые в книге, и страницы, на которых они упоминаются.

Значение, которое определяет местоположение записи таблицы, по стандарту CODASYL называется **ключом базы данных** (КБД). (В Oracle для этого используется термин *идентификатор строки*, RowID, в Postgres – *идентификатор кортежа*, TID). Значение КБД формируется системой при размещении записи в памяти и содержит информацию, позволяющую однозначно определить место размещения записи (преобразовать значение КБД в физический адрес записи). В качестве КБД может выступать, например, последовательный номер записи в файле или совокупность адреса страницы памяти и смещения от начала страницы.

Содержимое индекса

Значение атрибута	*КБД*
Белова	FA:00
Волков	F6:1E
Волкова	F6:00
Осипов	FA:2B
Поспелов	F6:31
Фридман	FA:1D

Пространство памяти

F6:00	Волкова	… (другие атрибуты)
F6:1E	Волков	…
F6:31	Поспелов	…
	…	
FA:00	Белова	…
FA:1D	Фридман	…
FA:2B	Осипов	…

Рис. 1. Пример индекса

Каждый индекс связан с определённой таблицей, но является внешним объектом по отношению к таблице и обычно хранится отдельно от неё в другом файле или отдельной области памяти.

В большинстве СУБД пустые значения атрибутов (NULL) не индексируются. Исключение составляет система Postgres.

Индексирование используется для ускорения доступа к записям по значению ключа и не влияет на размещение данных этой таблицы. Ускорение поиска данных через индекс обеспечивается за счёт:

1) упорядочивания значений индексируемого атрибута.
2) индекс обычно занимает меньше памяти, чем сама таблица, т.к. хранит не всю запись таблицы, а отдельные поля (чаще всего – одно поле). Поэтому система тратит меньше времени на чтение индекса, чем на чтение таблицы.

Индексы поддерживаются динамически, т.е. после обновления таблицы – добавления или удаления записей, а также модификации индексируемых полей, – индекс приводится в соответствие с последней версией данных таблицы. Обновление индекса, естественно, занимает некоторое время (иногда, очень большое), поэтому существование многих индексов может замедлить работу БД.

Обращение к записи таблицы через индексы осуществляется в два этапа: сначала СУБД считывает индекс в оперативную память (ОП) и находит в нём требуемое значение атрибута и соответствующий адрес записи (КБД), затем по этому адресу происходит обращение к внешнему запоминающему устройству. Индекс загружается в ОП целиком или хранится в ней постоянно во время работы с таблицей БД, если хватает объёма ОП.

1.2. Многоуровневые индексы

В современных СУБД самым распространенным типом индекса является **многоуровневый индекс** в виде сбалансированного дерева (B-дерево, balance tree). Дерево называется сбалансированным, потому что любой путь от корня к блокам-листьям имеет одинаковую длину. СУБД автоматически сортирует значения индексируемого атрибута, определяет необходимую глубину дерева (количество уровней) и размещает отсортированные данные в этой структуре. В качестве конкретного примера рассмотрим индексирование в виде B-дерева, которое используется в СУБД Oracle (Рис. 2).

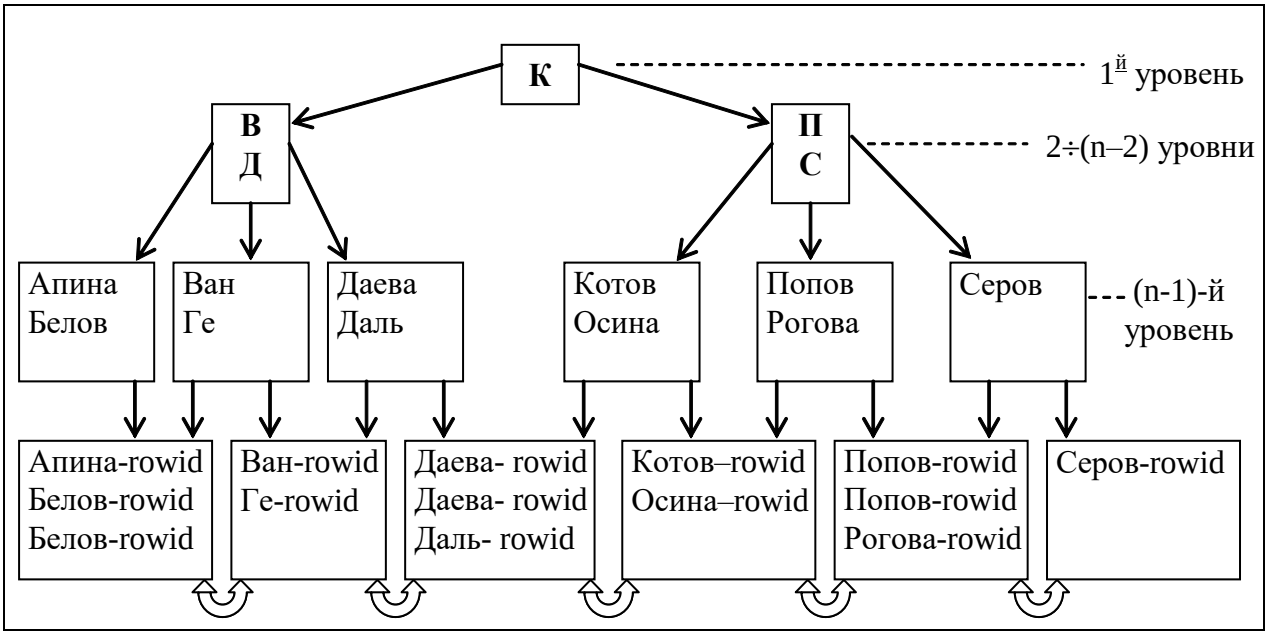


Рис. 2. Пример индекса в виде B-дерева (СУБД Oracle)

Верхние вершины индекса содержат автоматически вычисляемые значения, которые позволяют осуществлять поиск данных. Вершины 2÷(n-2) уровней содержат по два значения X_0 и X_k и три ссылки: левая ссылка ведет к поддереву со значениями

меньше X_0, правая ссылка ведет к поддереву со значениями больше X_k, средняя ссылка ведет к поддереву со значениями в диапазоне $[X_0, X_k]$. Предпоследний (n-1)-й уровень содержит значения индексируемого поля (атрибута) без повторов (т.е. каждое значение один раз). Количество значений в вершинах (блоках) (n-1)-го уровня зависит от размера самих значений и от размера блока (например, в Postgres размер блока фиксированный и всегда составляет 8К). Самый нижний n-й уровень – блоки-листья, которые содержат индексируемые значения и соответствующие идентификаторы записей RowID (row identification, КБД), используемые для нахождения самих записей. Для неуникальных индексов значения RowID в блоках-листьях индекса также отсортированы по возрастанию. Блоки-листья связаны между собой двунаправленными ссылками для ускорения поиска в диапазоне значений.

Рассмотрим поиск данных с помощью индекса (Рис. 2) на примере такого запроса:

```
select *
     from emp
     where lname='Даева';
```

При разборе запроса система ориентируется на условие (lname='Даева'). Из словаря-справочника БД она узнает, что по этому полю есть индекс. Данное условие (оператор =) позволяет ей запустить поиск по этому индексу, предварительно считав его из памяти. Корневая вершина (Рис. 2) содержит значение 'К' и указатели на верхнюю и нижнюю части индекса. Система сравнивает значение 'Даева' и 'К' и переходит к верхней части индекса (по левому указателю), т.к. 'Даева' < 'К'. Затем она сравнивает 'Даева' с 'В' и 'Д' и переходит направо, т.к. 'Даева' > 'Д'. На (n-1)-м уровне индекса она находит нужное значение 'Даева', переходит к блоку-листу и читает из него два RowID. Затем происходит обращение к диску по этим RowID и извлечение требуемых записей. Если бы в запросе было другое условие, например, (lname='Горин'), то система не нашла бы такого значения, не стала бы читать саму таблицу и выдала бы сообщение "Строки не найдены". Естественно, это выполняется быстрее, чем при полном чтении самой таблицы.

Другой пример:

```
select *
     from emp
     where lname<'Г';
```

Аналогично система ориентируется на условие в части **where**. Искомое значение 'Г' меньше 'К', поэтому система переходит по левому указателю. Значение 'Г' находится между 'В' и 'Д', поэтому она переходит по средней ссылке. На (n-1)-м уровне индекса она находит значение 'Ге' и берет крайнее значение, удовлетворяющее условию (значение 'Ван'); переходит к блоку-листу, читает из него нужный RowID. Потом переходит по ссылке между блоками-листьями налево и читает самый левый блок-лист, извлекая из него все RowID. Затем происходит обращение к диску по этим RowID и извлечение требуемых записей. Таким образом, она проходит от корня индекса к блокам-листьям один раз.

1.3. Общие сведения о создании индексов

В системах, поддерживающих язык SQL, индекс создаётся командой **create index**. Упрощенный синтаксис этой команды следующий:

```
CREATE INDEX <имя_индекса>
      ON <имя_таблицы>(<поле1> [, <поле2>,...])
      [<параметры>];
```

Имя индекса должно быть уникальным среди имён объектов БД. Если индекс составной, то входящие в него поля перечисляются через запятую. Необязательные <параметры> зависят от используемой СУБД.

Например, с помощью следующей команды можно создать составной индекс для таблицы *СОТРУДНИКИ* (EMP) по полям *Фамилия* (lname) и *Имя* (fname):

```
CREATE INDEX ind_emp_name ON emp(lname, fname);
```

СУБД **автоматически** создают индексы по первичным ключам и уникальным полям или комбинациям полей. По другим полям индексы должен построить пользователь (или разработчик БД). Но СУБД сама определяет для каждого выполняемого запроса, будет она пользоваться индексами или нет. В общем случае необходимыми и достаточными условиями использования индекса являются:

1) Необходимые: должно существовать условие (элемент запроса), позволяющий системе запустить алгоритм поиска по индексу. Чаще всего такими элементами являются:
   - условия в части **where** (**=, >, <, >=, <=, IN, BETWEEN, LIKE**);
   - order by или group by по полю (полям), по которому(-ым) есть индекс;
   - обращение в списке выбора только к полю (полям), по которому(-ым) есть индекс: в этом случае система может выбрать данные их индекса, вообще не обращаясь к таблице.
2) Достаточные: **по оценке СУБД** поиск данных по индексу должен занимать меньше времени, чем другие способы доступа, например, последовательное сканирование всей таблицы. Для этого система использует специальные алгоритмы оценки времени выполнения и статистику – информацию о распределении данных в индексированных полях.

Обратите внимание: условия != (<>), **not like, not in, not between** обычно подавляют использование индекса. Во-первых, для этих условий нет алгоритма, который мог бы осуществить поиск по индексу. Во-вторых, чаще всего эти условия обладают низкой селективностью. Исключение составляет предикат **not in**, который используется с подзапросом, но не для всех СУБД. Например, в системе Oracle следующий запрос будет выполнен с помощью индекса по полю **id** таблицы **job**:

```
SELECT * FROM staff
  WHERE id not in (select id from job);
```

Индексы повышают производительность запросов, которые выбирают относительно небольшое число строк из таблицы. Для определения целесообразности создания индекса нужно проанализировать запросы, обращённые к таблице, и распределение данных в индексируемых столбцах.

Система может воспользоваться индексом по определённому атрибуту, если в запросе на значение этого атрибута накладывается условие, например:

```
SELECT * FROM emp WHERE post = 'programmer';
```

Но даже при наличии такой возможности система не всегда обращается к индексу. Например, если запрос выбирает больше половины записей отношения, то извлечение данных через индекс потребует больше времени, чем последовательное чтение данных из таблицы. Это следует из того, что данные через индекс выбираются не в той последовательности, в которой они хранятся в памяти. Для подобных запросов построение индекса нецелесообразно.

Обращение к составному индексу возможно только в том случае, если в условиях выбора участвуют столбцы, представляющие собой лидирующую часть составного индекса. Если индекс, например, включает поля (X, Y, Z), то обращение к индексу может происходить тогда, когда в условии запроса есть поля XYZ, XY, XZ или X.

При создании индекса большое значение имеет понятие селективности. **Селективность** определяется процентом строк, имеющих одинаковое значение индексируемого столбца: чем выше этот процент, тем меньше селективность. Например, уникальный столбец обладает самой высокой селективностью, а столбец с двумя значениями – самой низкой. Но еще большее значение имеет селективность условия в запросе. Например, для уникального столбца B условие B=1000 обладает высокой селективностью, а B>1000 – низкой, если под него попадает 50% записей таблицы. Для столбца A с равномерным распределением значений от 1 до 100 условие A<=10 имеет достаточно высокую селективность (~10%), а условие A>10 имеет низкую селективность (~90%),

Удалить индекс можно с помощью команды **DROP INDEX**:

```
DROP INDEX <имя индекса>;
```

Удаление индекса не влияет на данные в таблице. Нельзя удалять индекс по первичному ключу, пока включено ограничение целостности **primary key**. Нецелесообразно удалять индекс по уникальному ключу, который система также создает автоматически. Даже если нет запросов, обращающихся по этому ключу к таблице, СУБД использует этот индекс для проверки ограничения уникальности, и его удаление повлечет за собой увеличение времени на выполнение этой проверки.

1.4. Анализ плана выполнения запроса

Итак, индексы создаются пользователем, но СУБД сама определяет для каждого выполняемого запроса, будет она пользоваться индексами или нет. Следовательно, пользователь должен уметь определять, пользуется система при выполнении запросов созданными индексами или нет. Наиболее часто используемые термины из плана выполнения запроса приведены в Приложении 1.

Рассмотрим несколько примеров анализа планов выполнения запросов, которые строит СУБД Oracle, для фрагмента БД "Проектная организация" (схемы таблиц приведены в Приложении 2).

1) Все данные о сотрудниках (Рис. 3):

```
select * from staff;
```

```
select * from staff;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			66	1	3	10 032		
TABLE ACCESS	FULL	STAFF	66	1	3	10 032		

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		IND_POST		POST	NONUNIQUE	VALID	NORMAL	NO

Рис. 3. План выполнения запроса (1)

Из плана выполнения (Рис. 3) видно, что индексы не используются: нет ни необходимых, ни достаточных условий. Система полностью читает таблицу **staff**. Ускорить выполнение такого запроса невозможно.

2) Все данные о сотрудниках 6-го отдела (Рис. 4):

```
select * from staff
   where depno=6;
```

```
select * from staff
where depno=6;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			15	1	3	2 280		
TABLE ACCESS	BY INDEX ROWID	STAFF	15	1	3	2 280		
INDEX	RANGE SCAN	IND_DEPNO	15	1	1			"DEPNO" = 6

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	IND_STAFF_POST		POST	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO	✓	DEPNO	NONUNIQUE	VALID	NORMAL	NO

Рис. 4. План выполнения запроса (2)

Из плана выполнения (Рис. 4) видно, что система использует индекс по полю **depno**: необходимое условие (**depno=6**), достаточное условие – небольшой процент выбираемых строк (ориентировочно менее 25%). Система читает индекс по полю

depno, ищет в нем значение 6, выбирает RowID, затем из таблицы **staff** читает записи по их RowID, что ускоряет выполнение запроса.

3) Все данные о программистах 6-го отдела (Рис. 4):

```
select * from staff
   where depno=6 and post like 'programm%';
```

```
select * from staff
where depno=6 and post like 'programm%';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	3	152		
TABLE ACCESS	BY INDEX ROWID	STAFF	1	1	3	152	"DEPNO" = 6	
INDEX	RANGE SCAN	IND_STAFF_POST	5	1	1		"POST" LIKE 'programm%'	"POST" LIKE 'programm%'

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	IND_STAFF_POST	✓	POST	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO		DEPNO	NONUNIQUE	VALID	NORMAL	NO

Рис. 5. План выполнения запроса (3)

Для запроса (3) система может воспользоваться двумя индексами: по полю **depno** и по полю **post**. Для обращения к одной таблице система чаще всего выбирает один индекс – с наиболее селективным условием. В данном случае она использует индекс по полю **post** (Рис. 4). Система читает индекс по полю **post**, ищет в нем значения, которые начинаются с 'programm', выбирает RowID, затем из таблицы **staff** читает записи по их RowID и проверяет для них второе условие (**depno=6**).

4) Все данные о программистах 5-го отдела (

```
select * from staff
where depno=5 and post like '%programm%';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	2	152		
TABLE ACCESS	BY INDEX ROWID	STAFF	1	1	2	152	"POST" LIKE '%programm%'	
INDEX	RANGE SCAN	IND_DEPNO	8	1	1			"DEPNO" = 5

* Unindexed columns are shown in red

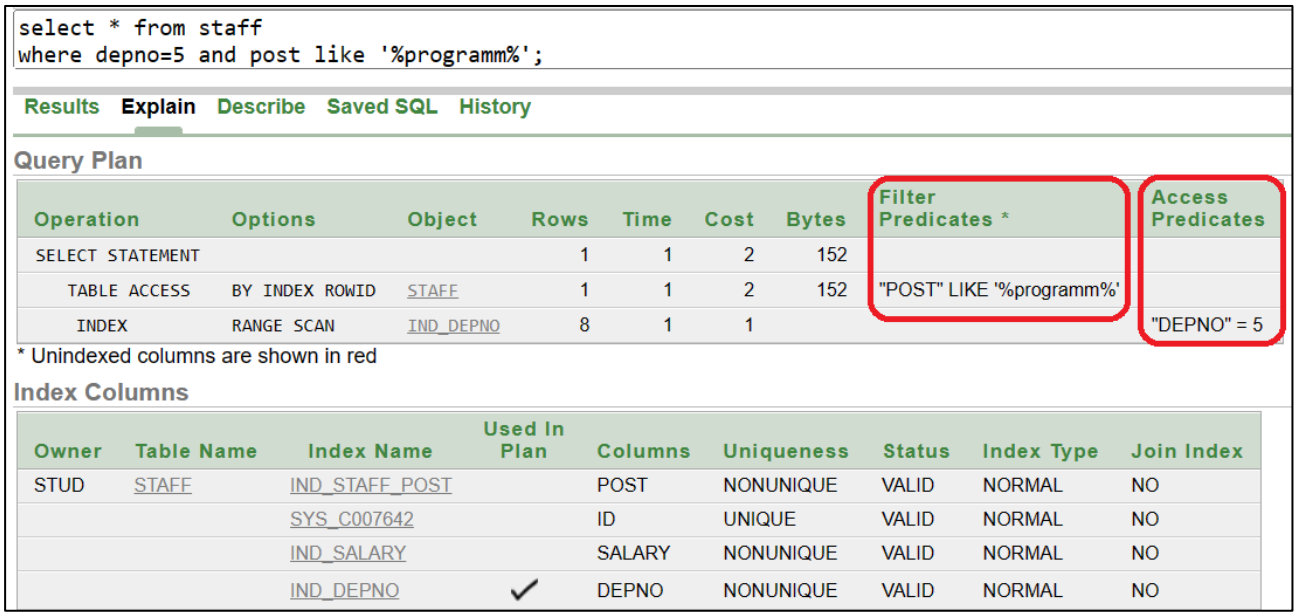
Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	IND_STAFF_POST		POST	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO	✓	DEPNO	NONUNIQUE	VALID	NORMAL	NO

5) Рис. 6):

```
select * from staff
```

```
where depno=5 and post like '%programm%';
```



```
select * from staff
where depno=5 and post like '%programm%';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	2	152		
TABLE ACCESS	BY INDEX ROWID	STAFF	1	1	2	152	"POST" LIKE '%programm%'	
INDEX	RANGE SCAN	IND_DEPNO	8	1	1			"DEPNO" = 5

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	IND_STAFF_POST		POST	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO	✓	DEPNO	NONUNIQUE	VALID	NORMAL	NO

Рис. 6. План выполнения запроса (4)

В запросе (4) учтено, что слово "программист" может быть в названии должности не первым. Теперь система не может воспользоваться индексом по полю **post**, т.к. в условии (**post like '%programm%'**) неопределенная лидирующая часть, и в системе нет такого алгоритма поиска по индексу (нет необходимого условия поиска по индексу). В данном случае она использует индекс по полю **depno** (Рис. 6). Система читает индекс по полю **depno**, ищет в нем значение 5, выбирает RowID, затем из таблицы **staff** читает записи по их RowID и проверяет для них второе условие. Это видно на Рис. 6: условие (**depno = 5**) является предикатом доступа, а условие (**post like '%programm%'**) – предикатом фильтрации.

6) Все данные о сотрудниках с идентификаторами меньше 30 (Рис. 7):

```
select * from staff
  where id < 30;
```

```
select * from staff
where id<30;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			9	1	2	1 368		
TABLE ACCESS	BY INDEX ROWID	STAFF	9	1	2	1 368		
INDEX	RANGE SCAN	SYS_C007642	9	1	1			"ID"<30

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	SYS_C007642	✓	ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		IND_POST		POST	NONUNIQUE	VALID	NORMAL	NO

Рис. 7. План выполнения запроса (5)

Для запроса (5) система пользуется индексом по полю **id**: для этого есть и необходимые, и достаточные условия (Рис. 7).

7) Все данные о сотрудниках с идентификаторами не менее 30 (Рис. 8):

```
select * from staff
  where id>=30;
```

Для запроса (6) система не пользуется индексом по полю **id** (Рис. 8): есть необходимое условие, но нет достаточного. Условие (**id>=30**) имеет низкую селективность: по нему нужно прочитать из таблицы более половины строк. Это неэффективно делать через индекс, поэтому система выбирает полное чтение таблицы (FULL).

```
select * from staff
where id>=30;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			57	1	3	8 664		
TABLE ACCESS	FULL	STAFF	57	1	3	8 664	"ID"> = 30	

* Unindexed columns are shown in red

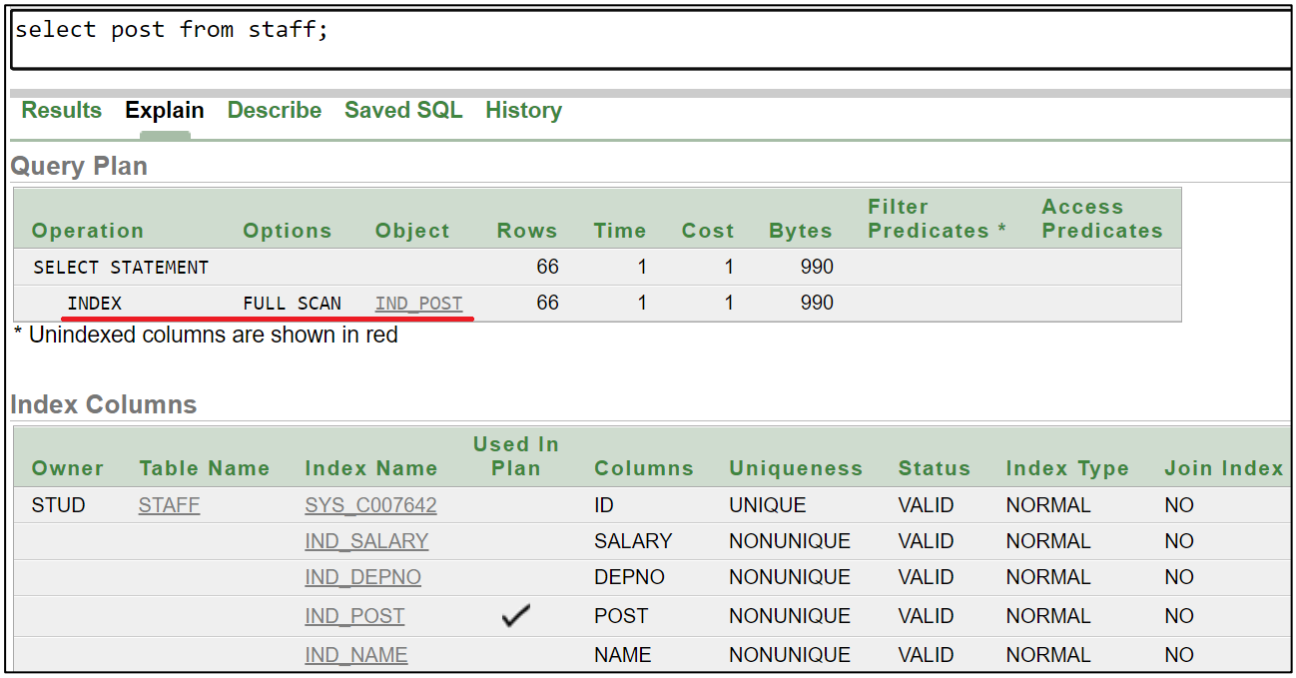
Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		IND_POST		POST	NONUNIQUE	VALID	NORMAL	NO
		IND_NAME		NAME	NONUNIQUE	VALID	NORMAL	NO

Рис. 8. План выполнения запроса (6)

8) Перечень всех должностей сотрудников (Рис. 9):

```
select post from staff;
```



```
select post from staff;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			66	1	1	990		
INDEX	FULL SCAN	IND_POST	66	1	1	990		

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	STAFF	SYS_C007642		ID	UNIQUE	VALID	NORMAL	NO
		IND_SALARY		SALARY	NONUNIQUE	VALID	NORMAL	NO
		IND_DEPNO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		IND_POST	✓	POST	NONUNIQUE	VALID	NORMAL	NO
		IND_NAME		NAME	NONUNIQUE	VALID	NORMAL	NO

Рис. 9. План выполнения запроса (7)

На первый взгляд, может показаться удивительным, что для запроса (7) система пользуется индексом по полю **post** (Рис. 9): ведь этот запрос не содержит условия выбора. Но в данном случае роль необходимого условия играет список выбора: он содержит единственное поле, и для этого поля есть индекс. Таким образом, система вообще не обращается к таблице, а читает только индекс (INDEX FULL SCAN). Так как индекс занимает обычно меньше памяти, чем таблица, и в индексе есть все нужные значения этого поля, то это эффективно делать через индекс.

9) Данные о проектах и ролях участников этих проектов (Рис. 9):

```
select *
   from project p, job j
   where p.pid = j.pid;
```

Это запрос на соединение таблиц с условием соединения в части `where`. Таблицы **project** и **job** связаны внешним ключом, при этом есть индексы на первичный ключ **p.pid** (создан автоматически) и на внешний ключ **j.pid**. Соединение строится на основе декартова произведения таблиц, но это очень ресурсозатратная операция. Ее можно ускорить с помощью использования индекса. Но условие **p.pid=j.pid** без предварительных действий не дает возможность использовать ни один из индексов, потому что система не знает, какое значение надо искать в индексе. Значит, надо предварительно прочитать данные из одной таблицы.

В данном случае (Рис. 10) система начинает читать таблицу **project** (TABLE ACCESS FULL). Прочитав одну запись, она получает значение **p.pid**, и теперь может найти его в индексе по внешнему ключу **j.pid**, извлечь RowID и считать нужные записи из таблицы **job**. Таким образом, система соединяет каждую прочитанную

запись из **project** со всеми записями таблицы **job**, для которых выполняется условие соединения, и ей не надо проверять это условие для других записей из **job**.

```
select *
from job j, project p
where p.pid = j.pid;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			29	1	4	2 639		
TABLE ACCESS	BY INDEX ROWID	JOB	3	1	1	81		
NESTED LOOPS			29	1	4	2 639		
TABLE ACCESS	FULL	PROJECT	9	1	3	576		
INDEX	RANGE SCAN	IND_PID	5	1	0			"P"."PID" = "J"."PID"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	JOB	IND_JOB_ID		ID	NONUNIQUE	VALID	NORMAL	NO
		IND_PID	✓	PID	NONUNIQUE	VALID	NORMAL	NO
	PROJECT	IND_DEPNO_PRO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007653		PID	UNIQUE	VALID	NORMAL	NO

Рис. 10. План выполнения запроса (8)

Обратите внимание, если запрос (8) переписать через оператор **JOIN**, план выполнения **не изменится** (Рис. 11). Это просто другая форма записи операции соединения, но с точки зрения эффективности выполнения это одинаковые запросы.

```
select *
from project p JOIN job j ON p.pid = j.pid;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			29	1	4	2 639		
TABLE ACCESS	BY INDEX ROWID	JOB	3	1	1	81		
NESTED LOOPS			29	1	4	2 639		
TABLE ACCESS	FULL	PROJECT	9	1	3	576		
INDEX	RANGE SCAN	IND_PID	5	1	0			"J"."PID" = "P"."PID"

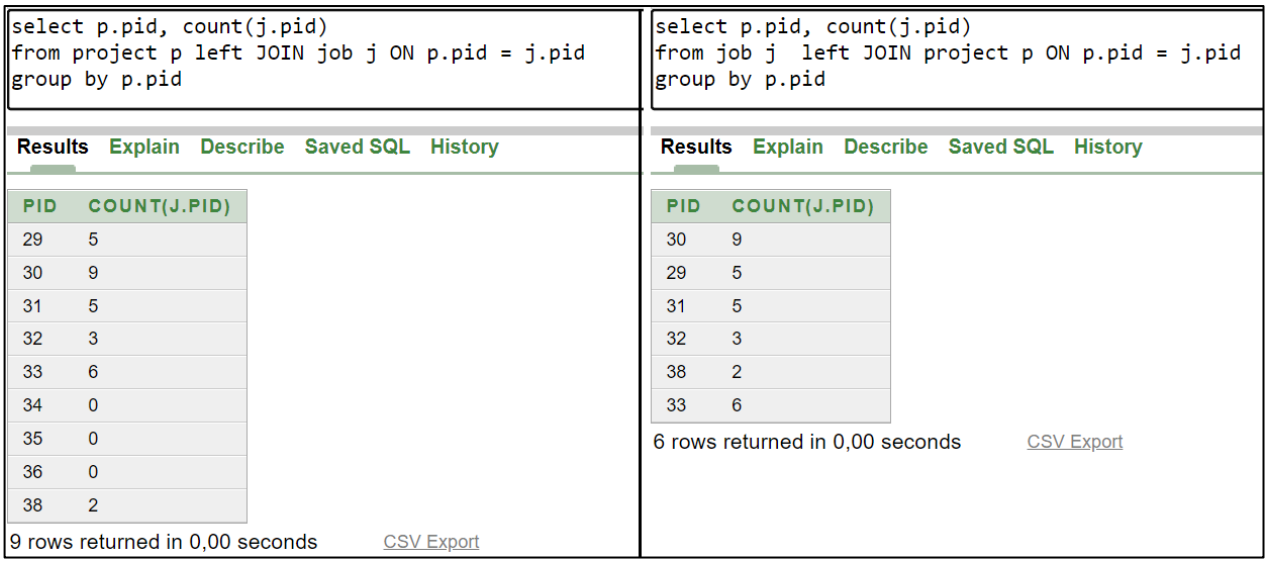
* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
STUD	JOB	IND_JOB_ID		ID	NONUNIQUE	VALID	NORMAL	NO
		IND_PID	✓	PID	NONUNIQUE	VALID	NORMAL	NO
	PROJECT	IND_DEPNO_PRO		DEPNO	NONUNIQUE	VALID	NORMAL	NO
		SYS_C007653		PID	UNIQUE	VALID	NORMAL	NO

Рис. 11. План выполнения запроса на соединение через `JOIN`

Другое дело, если заменить в запросе на соединение таблиц условие соединения в части `where` или обычный **`JOIN`** (**`INNER JOIN`**) на **`LEFT JOIN`**. Открытое соединение вычисляется не так, как закрытое: система берет из левой таблицы все записи, даже если для них не выполняется условие соединения. Например, на Рис. 12,а слева стоит родительская таблица **`project`**, и в результате система выводит те проекты, в которых нет участников (count = 0). А на Рис. 12,б слева стоит подчиненная таблица **`job`**, и результат идентичен **`INNER JOIN`**.



```
select p.pid, count(j.pid)
from project p left JOIN job j ON p.pid = j.pid
group by p.pid
```

Results Explain Describe Saved SQL History

PID	COUNT(J.PID)
29	5
30	9
31	5
32	3
33	6
34	0
35	0
36	0
38	2

9 rows returned in 0,00 seconds CSV Export

```
select p.pid, count(j.pid)
from job j  left JOIN project p ON p.pid = j.pid
group by p.pid
```

Results Explain Describe Saved SQL History

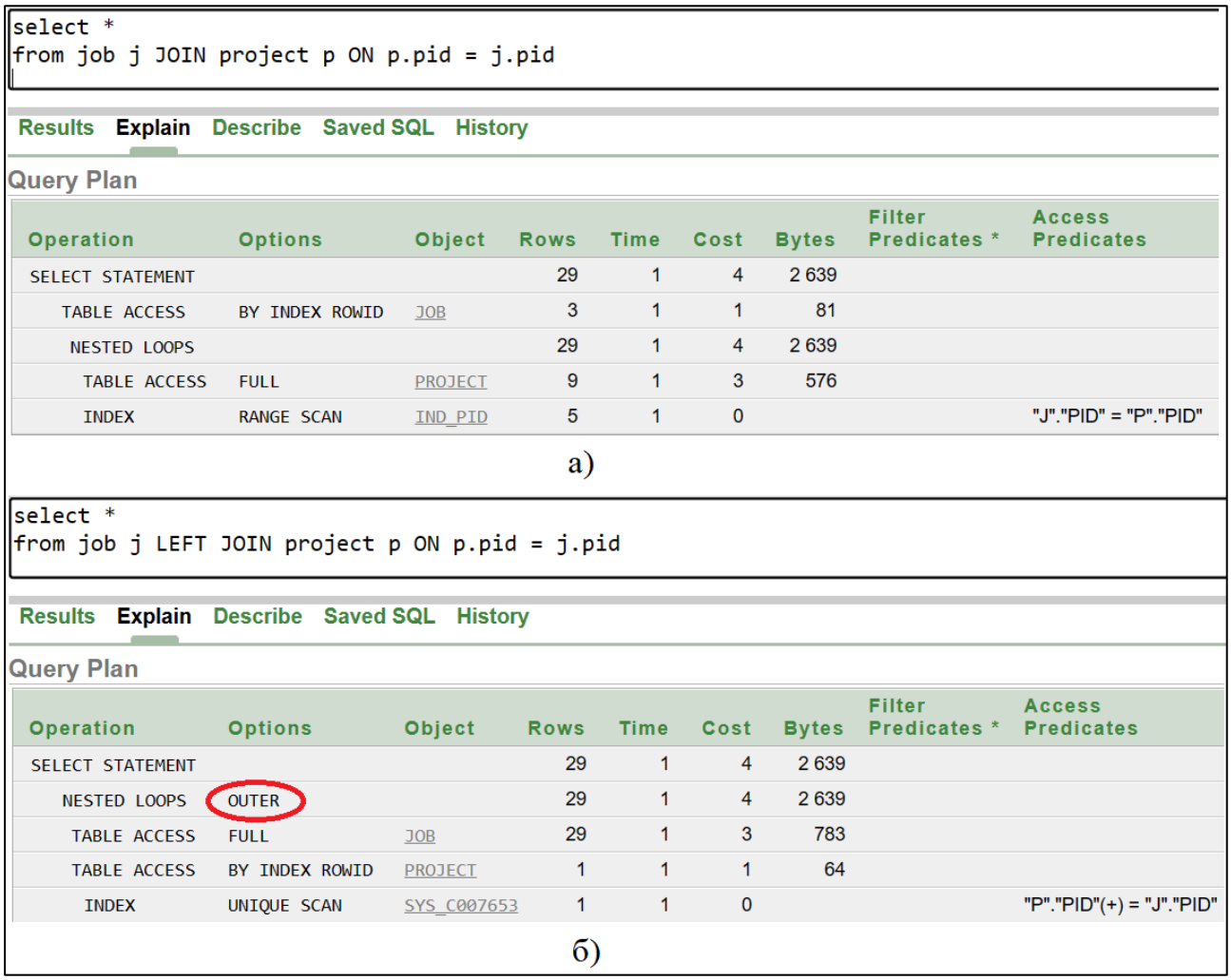
PID	COUNT(J.PID)
30	9
29	5
31	5
32	3
38	2
33	6

6 rows returned in 0,00 seconds CSV Export

а) б)

Рис. 12. Результаты выполнения запросов на левое открытое соединение

Но использовать **`LEFT JOIN`** вместо **`JOIN`** (**`INNER JOIN`**) – это очень плохой вариант. Открытое соединение не может быть выполнено быстрее закрытого, а часто вычисляется дольше. Это объясняется просто: система не может просто взять запись из левой таблицы и положить ее в результат; ей все равно надо проверить наличие или отсутствие соответствующей записи в правой таблице. Но план выполнения при этом поменяется. Для выполнения запроса с закрытым соединением система выбрала полное чтение таблицы **`project`**, обращение по индексу для внешнего ключа и чтение записей таблицы **`job`** по RowID (Рис. 13,а), т.е. она первой считала таблицу, стоящую второй в части **`from`**.



```
select *
from job j JOIN project p ON p.pid = j.pid
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			29	1	4	2 639		
TABLE ACCESS	BY INDEX ROWID	JOB	3	1	1	81		
NESTED LOOPS			29	1	4	2 639		
TABLE ACCESS	FULL	PROJECT	9	1	3	576		
INDEX	RANGE SCAN	IND_PID	5	1	0			"J"."PID" = "P"."PID"

а)

```
select *
from job j LEFT JOIN project p ON p.pid = j.pid
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			29	1	4	2 639		
NESTED LOOPS	OUTER		29	1	4	2 639		
TABLE ACCESS	FULL	JOB	29	1	3	783		
TABLE ACCESS	BY INDEX ROWID	PROJECT	1	1	1	64		
INDEX	UNIQUE SCAN	SYS_C007653	1	1	0			"P"."PID"(+) = "J"."PID"

б)

Рис. 13. Сравнение планов выполнения запросов на закрытое (а) и открытое (б) соединения

А для открытого соединения система не может изменить порядок обработки таблиц: она выбрала полное чтение таблицы **`job`**, обращение по индексу для первичного ключа и чтение записей таблицы **`project`** по RowID (Рис. 13,б), что привело к увеличению количества обрабатываемых строк (колонка Rows) и количества байтов (колонка Bytes), хотя сумма оценок (колонка Cost) не поменялась.

Вывод из этого очень простой: не следует использовать открытое соединение там, где можно обойтись обычным закрытым соединением (**`inner join`**).

1.5. Создание индексов по набору запросов

Как и любой метод, индексирование имеет достоинства и недостатки. К достоинствам относится возможное ускорение выполнения запросов; к недостаткам – дополнительный расход памяти на хранение индексов и затраты на изменение индексов при выполнении команд **`insert, update, delete`**. Индексы создаются по результатам анализа запросов, которые обращаются к базе данных. Набор выполняемых запросов определяется потребностями пользователей предметной области. Для большинства предметных областей он примерно на 90-95% определен и известен до начала эксплуатации БД. При правильном проектировании базы данных эти запросы оформляются как представления или процедуры.

Для определения набора необходимых индексов есть несколько очевидных правил, которым надо следовать:

1. В первую очередь, надо анализировать те запросы, которые создают наибольшую нагрузку на БД: обычно здесь работает правило 20/80, то есть примерно 20% самых часто используемых или объемных запросов создают 80% нагрузки.
2. По возможности, следует создавать такие индексы, которые система сможет использовать для выполнения нескольких запросов. Для этого хорошо подходят составные индексы, но они могут потребовать внесения изменений в запросы.

Выбор столбцов для индекса определяется следующими соображениями:

- В первую очередь выбираются столбцы, которые часто встречаются в условиях поиска.
- Стоит индексировать столбцы, которые используются для соединения таблиц или являются внешними ключами. Такой индекс позволяет системе не строить декартово произведение для запросов на соединение таблиц.
- Нецелесообразно индексировать столбцы с низкой селективностью. Исключения для низкой селективности составляют случаи, при которых выборка производится только по редко встречающимся значениям.
- Не индексируются столбцы, которые часто обновляются, т.к. команды обновления ведут к потере времени на обновление индекса.
- Не индексируются столбцы, которые часто используются как аргументы выражений или функций: как правило, это не позволяет использовать индекс.

В некоторых случаях использование составного индекса предпочтительнее, чем одиночного, а именно:

- Несколько столбцов с низкой селективностью в комбинации друг с другом могут дать гораздо более высокую селективность.
- Если в запросах часто используются только столбцы, участвующие в индексе, система может вообще не обращаться к таблице для поиска данных.

При создании индексов нужно учитывать интенсивность запросов на чтение и на изменение данных в таблице и объем извлекаемых данных. Если таблица редко меняется, но часто запрашивается, для нее можно создать индексы на все используемые при поиске поля или комбинации полей. Например, пусть есть три запроса:

```
select * from Tab where a = 10 and b > Y;
select * from Tab where b = Y and c IN (1, 2, 4);
select * from Tab where a < X and c = Z;
```

Тогда имеет смысл создать индекс на три поля (**a, b, c**) и добавить во второй запрос условие на поле **a**, которое позволит системе использовать этот индекс, задавая условие на лидирующую часть. Например, если мы знаем, что это числовое поле и оно не может иметь отрицательных значений, то условие может быть **a>0**.

Если таблица часто меняется, то нужно создавать только минимально необходимое количество индексов, чтобы время на изменение самих индексов не превысило выигрыш от более быстрого чтения данных с помощью индекса.

В общем, для всех критически важных запросов должны быть созданы индексы, а запросы должны быть написаны так, чтобы система могла их использовать. Количество индексов должно быть относительно небольшим, чтобы общая производительность системы с индексами была выше, чем без них.

Рассмотрим процесс создания набора индексов на примере предметной области "Кинотеатр". Схема БД и команды создания таблиц приведены в Приложении 3. Ниже приведены некоторые запросы для этой БД:

A. Все фильмы-комедии с рейтингом выше 8:

```
SELECT *
FROM films
WHERE f_genre = 'comedy' and f_rating > 8;
```

B. Все фильмы-драмы и мелодрамы с возрастным ограничением 16 и выше:

```
SELECT *
FROM films
WHERE f_genre in ('drama', 'melodrama') and f_age >= 16;
```

C. Фильмы, выпущенные за последние пять лет в США:

```
SELECT *
FROM films
WHERE f_country = 'USA' and f_year>=to_number(to_char(current_date, 'yyyy'))-4;
```

D. Все фильмы, названия которых начинаются с 'La':

```
SELECT *
FROM films
WHERE f_name like 'La%';
```

E. Все вечерние сеансы на сегодня:

```
SELECT f.f_name, trunc(s.s_date), to_char(s.s_date, 'hh24:mi'), r_name
FROM sessions s JOIN films f ON s.s_film=f.f_id join halls on r_num = s_hall
WHERE trunc(s.s_date)=trunc(sysdate) and to_char(s_date, 'hh24')>'15';
```

F. Все доступные вечерние сеансы на завтра на фильм 'Once in Hollywood':

```
SELECT f.f_name, trunc(s.s_date), to_char(s.s_date, 'hh24:mi'), r_name
FROM sessions s JOIN films f ON s.s_film=f.f_id join halls on r_num = s_hall
WHERE trunc(s.s_date)=trunc(sysdate)+1
 and f_name = 'Once in Hollywood' AND to_char(s_date, 'hh24')>'18' and
EXISTS(SELECT * FROM tickets t WHERE t.t_session=s.s_num AND t.t_status='free');
```

G. Фильмы, которые находятся в прокате в ближайшее время (например, 2 недели):

```
SELECT f_name, f_director, f_age, f_duration, f_rating, f_year, f_country
FROM films f
WHERE EXISTS(SELECT * FROM sessions s
      WHERE (f.f_id=s.s_film) AND (s.s_date BETWEEN current_date
                                   AND current_date+14));
```

H. Все доступные на данный момент билеты на сегодня на фильм 'Once in Hollywood' (есть свободное место и подходит по дате и времени):

```
SELECT f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall, t_row, t_place
FROM tickets t JOIN sessions s ON t.t_session=s.s_num
      JOIN films f ON s.s_film=f.f_id
WHERE trunc(s.s_date)=trunc(sysdate)+1 and s.s_date>sysdate AND t_status='free'
      and f_name = 'Once in Hollywood';
```

I. Количество доступных билетов на сегодня на данный момент:

```
SELECT f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall, count(*)
FROM tickets t JOIN sessions s ON t.t_session=s.s_num
      JOIN films f ON s.s_film=f.f_id JOIN halls r ON r.r_num=s.s_hall
WHERE trunc(s.s_date)=trunc(sysdate) and s.s_date>sysdate AND t_status='free'
group by f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall;
```

Очевидно, что самыми часто используемыми запросами будут запросы к таблицам "Фильмы" и "Сеансы" с указанием дополнительных фильтров: название фильма, жанр, рейтинг, возрастное ограничение, а также дата и время сеансов. Таблица "Фильмы" меняется редко, но часто запрашивается: по ней нужно создать все необходимые индексы. Таблица "Сеансы" меняется чаще, но и запрашивается часто: по ней тоже нужно создать все необходимые индексы. Таблица "Билеты" меняется еще чаще, поэтому по ней не следует создавать много индексов, только самые необходимые. Следовательно, можно предложить следующий набор индексов:

1. Составной индекс для полей (жанр, рейтинг, страна, возрастное ограничение) таблицы Фильмы. С помощью этого индекса система сможет быстрее находить фильмы для запросов (1), (2), (3).

```
create index ind_film_genre4 on films(f_genre,f_rating,f_country,f_age);
```

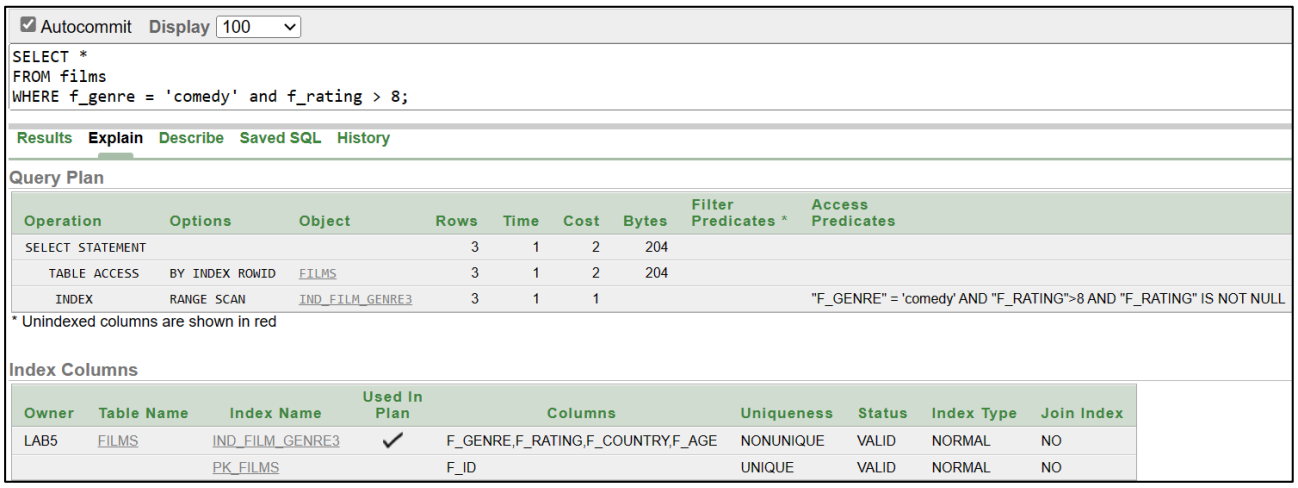


Рис. 14. План выполнения запроса (A)

Из Рис. 14 видно, что система обращается к созданному индексу, используя условия (**`f_genre='comedy' and f_rating>8`**) как условие доступа (Access predicates).
Для запроса (B) система использует индекс (Рис. 15): в нем есть условие на первое поле составного индекса, и этого достаточно, чтобы система им воспользовалась.

```
SELECT *
FROM films
WHERE f_genre in ('drama', 'melodrama') and f_age >= 16;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	2	68		
INLIST ITERATOR								
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	2	68		
INDEX	RANGE SCAN	IND_FILM_GENRE3	1	1	1		"F_AGE"> = 16	("F_GENRE" = 'drama' OR "F_GENRE" = 'melodrama') AND "F_AGE"> = 16

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3	✓	F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO

Рис. 15. План выполнения запроса (B)

```
SELECT *
FROM films
WHERE f_country = 'USA' and f_year>=to_number(to_char(current_date, 'yyyy'))-4;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	3	68		
TABLE ACCESS	FULL	FILMS	1	1	3	68	"F_COUNTRY" = 'USA' AND "F_YEAR"> = TO_NUMBER(TO_CHAR(CURRENT_DATE,'yyyy'))-4	

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO

Рис. 16. План выполнения запроса (C)

Для запроса (C) система индекс не использует (Рис. 16), потому что в нем нет условия на первое поле составного индекса. Добавим условие на это поле:

```
SELECT * FROM films
WHERE f_country = 'USA' and f_year>=to_number(to_char(current_date, 'yyyy'))-4
      and f_genre > '1';
```

Условие должно быть таким, чтобы не влиять на результат: не исключать из него ни одной записи. Теперь система может воспользоваться созданным индексом (Рис. 17).

```
SELECT *
FROM films
WHERE f_country = 'USA' and f_year>=to_number(to_char(current_date, 'yyyy'))-4
and f_genre > '1';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	2	68		
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	2	68	"F_YEAR"> = TO_NUMBER(TO_CHAR(CURRENT_DATE,'yyyy'))-4	
INDEX	RANGE SCAN	IND_FILM_GENRE3	6	1	1		"F_COUNTRY" = 'USA'	"F_GENRE">'1' AND "F_COUNTRY" = 'USA' AND "F_GENRE" IS NOT NULL

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3	✓	F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO

Рис. 17. План выполнения модифицированного запроса (C)

2. Индекс для названия фильма. С помощью этого индекса система сможет быстро находить фильмы по названию – запросы (D), (F), (H).

```
create index ind_film_name on films(f_name);
```

```
SELECT * FROM films
where f_name like 'La%';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	2	68		
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	2	68		
INDEX	RANGE SCAN	IND_FILM_NAME	1	1	1		"F_NAME" LIKE 'La%'	"F_NAME" LIKE 'La%'

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME	✓	F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO

Рис. 18. План выполнения запроса (D)

3. Индекс для таблицы Сеансы по внешним ключам. Так как эта таблица имеет два внешних ключа – на Залы и на Фильмы, и в большинстве запросов она соединяется с обеими таблицами, но чаще – с таблицей Фильмы, то имеет смысл также создать составной индекс, добавив в него еще и дату (для запросов (D), (E), (G), (H), (I)). Первым здесь будет внешний ключ **`s_film`**, т.к. с таблицей Фильмы сеансы соединяются во всех перечисленных запросах.

```
create index ind_session_film3 on sessions(s_film, s_hall, s_date);
```

```
SELECT f.f_name, trunc(s.s_date), to_char(s.s_date, 'hh24:mi'), r_name
FROM sessions s JOIN films f ON s.s_film=f.f_id join halls on r_num = s_hall
WHERE trunc(s.s_date)=trunc(sysdate) and to_char(s_date, 'hh24')>'15';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	3	41		
NESTED LOOPS			1	1	3	41		
NESTED LOOPS			1	1	2	25		
INDEX	FULL SCAN	IND_SESSION_FILM3	1	1	1	14	TO_CHAR(INTERNAL_FUNCTION("S"."S_DATE"),'hh24')>'15' AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)	
TABLE ACCESS	BY INDEX ROWID	HALLS	1	1	1	11		
INDEX	UNIQUE SCAN	PK_ROOMS	1	1	0			"S"."S_HALL" = "R_NUM"
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	1	16		
INDEX	UNIQUE SCAN	PK_FILMS	1	1	0			"S"."S_FILM" = "F"."F_ID"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME		F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS	✓	F_ID	UNIQUE	VALID	NORMAL	NO
	HALLS	PK_ROOMS	✓	R_NUM	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM3	✓	S_FILM,S_HALL,S_DATE	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS		S_NUM	UNIQUE	VALID	NORMAL	NO

Рис. 19. План выполнения запроса (E)

```
SELECT f_name, f_director, f_age, f_duration, f_rating, f_year, f_country
FROM films f
WHERE EXISTS(SELECT * FROM sessions s WHERE (f.f_id=s.s_film) AND (s.s_date BETWEEN current_date AND current_date+14));
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	3	63		
FILTER							CURRENT_DATE< = CURRENT_DATE+14	
NESTED LOOPS			1	1	3	63		
SORT	UNIQUE		1	1	1	11		
INDEX	FULL SCAN	IND_SESSION_FILM3	1	1	1	11	"S"."S_DATE"> = CURRENT_DATE AND "S"."S_DATE"< = CURRENT_DATE+14	"S"."S_DATE"> = CURRENT_DATE AND "S"."S_DATE"< = CURRENT_DATE+14
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	1	52		
INDEX	UNIQUE SCAN	PK_FILMS	1	1	0			"F"."F_ID" = "S"."S_FILM"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME		F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS	✓	F_ID	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM3	✓	S_FILM,S_HALL,S_DATE	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS		S_NUM	UNIQUE	VALID	NORMAL	NO

Рис. 20. План выполнения запроса (G)

4. Индекс для внешнего ключа таблицы Билеты:
   `create index ind_ticket_session on tickets(t_session);`
   С помощью этого индекса система сможет соединить таблицы Билеты и Сеансы для запросов (F), (H) и (I).

```
SELECT f.f_name, trunc(s.s_date), to_char(s.s_date, 'hh24:mi'), r_name
FROM sessions s JOIN films f ON s.s_film=f.f_id join halls on r_num = s_hall
WHERE trunc(s.s_date)=trunc(sysdate)+1
 and f_name = 'Once in Hollywood' AND to_char(s_date, 'hh24')>'18' and
EXISTS(SELECT * FROM tickets t WHERE t.t_session=s.s_num AND t.t_status='free');
```

Results Explain Describe Saved SQL History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			1	1	7	54		
NESTED LOOPS	SEMI		1	1	7	54		
NESTED LOOPS			1	1	5	44		
NESTED LOOPS			1	1	4	33		
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	2	16		
INDEX	RANGE SCAN	IND_FILM_NAME	1	1	1			"F"."F_NAME" = 'Once in Hollywood'
TABLE ACCESS	BY INDEX ROWID	SESSIONS	1	1	2	17		
INDEX	RANGE SCAN	IND_SESSION_FILM3	1	1	1		TO_CHAR(INTERNAL_FUNCTION("S"."S_DATE"),'hh24')>'18' AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)+1	"S"."S_FILM" = "F"."F_ID"
TABLE ACCESS	BY INDEX ROWID	HALLS	1	1	1	11		
INDEX	UNIQUE SCAN	PK_ROOMS	1	1	0			"S"."S_HALL" = "R_NUM"
TABLE ACCESS	BY INDEX ROWID	TICKETS	1 551	1	2	15 510	"T""T_STATUS" = 'free'	
INDEX	RANGE SCAN	IND_TICKET_SESSION	1	1	1			"T"."T_SESSION" = "S"."S_NUM"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME	✓	F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO
	HALLS	PK_ROOMS	✓	R_NUM	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM3	✓	S_FILM,S_HALL,S_DATE	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS		S_NUM	UNIQUE	VALID	NORMAL	NO
	TICKETS	IND_TICKET_SESSION	✓	T_SESSION	NONUNIQUE	VALID	NORMAL	NO
		PK_TICKETS		T_ID	UNIQUE	VALID	NORMAL	NO

Рис. 21. План выполнения запроса (F)

Благодаря созданным индексам ни одна таблица из четырех не читается полностью, ко всем идет обращение через индексы (Рис. 21). Единственное используемое при поиске поле, которое не проиндексировано, это поле **`t_status`** таблицы Билеты (оно выделено красным в плане выполнения). Но индекс для поля "статус билета" таблицы Билеты создавать не следует, т.к. это поле часто меняется, что будет приводить к большим затратам на изменение индекса. Все равно обращение к таблице Билеты происходит для определенного сеанса, поэтому главное – ускорить соединение этих таблиц с помощью индекса по внешнему ключу.

Аналогично мы можем видеть использование созданных индексов для запроса (H) (Рис. 22) и запроса (I) (Рис. 23).

```
SELECT f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall, t_row, t_place
FROM tickets t JOIN sessions s ON t.t_session=s.s_num JOIN films f ON s.s_film=f.f_id
WHERE trunc(s.s_date)=trunc(sysdate)+1 and s.s_date>sysdate AND t_status='free' and f_name = 'Once in Hollywood';
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			6	1	9	312		
TABLE ACCESS	BY INDEX ROWID	TICKETS	148	1	5	2 368	"T""T_STATUS" = 'free'	
NESTED LOOPS			6	1	9	312		
NESTED LOOPS			1	1	4	36		
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	2	19		
INDEX	RANGE SCAN	IND_FILM_NAME	1	1	1			"F"."F_NAME" = 'Once in Hollywood'
TABLE ACCESS	BY INDEX ROWID	SESSIONS	1	1	2	17		
INDEX	RANGE SCAN	IND_SESSION_FILM3	1	1	1		"S"."S_DATE">SYSDATE@! AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)+1	"S"."S_FILM" = "F"."F_ID" AND "S"."S_DATE">SYSDATE@!
INDEX	RANGE SCAN	IND_TICKET_SESSION	198	1	1			"T"."T_SESSION" = "S"."S_NUM"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME	✓	F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS		F_ID	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM3	✓	S_FILM,S_HALL,S_DATE	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS		S_NUM	UNIQUE	VALID	NORMAL	NO
	TICKETS	IND_TICKET_SESSION	✓	T_SESSION	NONUNIQUE	VALID	NORMAL	NO
		PK_TICKETS		T_ID	UNIQUE	VALID	NORMAL	NO

Рис. 22. План выполнения запроса (H)

Из плана выполнения для запроса (I) (Рис. 23) видно, что система строит временное представление (**`VIEW index$_join$_002`**), соединяя индекс по первичному ключу **`s_num`** таблицы Сессии и созданный составной индекс **`ind_session_film3`**. Если добавить в этот индекс поле **`s_num`**, то системе не придется строить временное представление, и время выполнения запроса уменьшится: сумма значений столбца **Cost** на Рис. 23 равна 48, а после изменения индекса – 34 (Рис. 24).

Такой составной индекс положительно повлияет на время выполнения запроса (F) – 21 против 33, и не изменит оценку времени выполнения запроса (H).

```
SELECT f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi') time, s.s_hall, count(*) cnt
FROM tickets t JOIN sessions s ON t.t_session=s.s_num JOIN films f ON s.s_film=f.f_id JOIN halls r ON r.r_num=s.s_hall
WHERE trunc(s.s_date)=trunc(sysdate) and s.s_date>sysdate AND t_status='free'
group by f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			6	1	10	294		
HASH	GROUP BY		6	1	10	294		
TABLE ACCESS	BY INDEX ROWID	TICKETS	148	1	5	1 480	"T""T_STATUS" = 'free'	
NESTED LOOPS			6	1	9	294		
NESTED LOOPS			1	1	4	39		
NESTED LOOPS			1	1	3	20		
VIEW		index$_join$_002	1	1	3	17	"S"."S_DATE">SYSDATE@! AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)	
HASH JOIN								ROWID = ROWID
INDEX	FAST FULL SCAN	IND_SESSION_FILM3	1	1	1	17	"S"."S_DATE">SYSDATE@! AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)	
INDEX	FAST FULL SCAN	PK_SESSIONS	1	1	1	17		
INDEX	UNIQUE SCAN	PK_ROOMS	1	1	0	3		"S"."S_HALL" = "R"."R_NUM"
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	1	19		
INDEX	UNIQUE SCAN	PK_FILMS	1	1	0			"S"."S_FILM" = "F"."F_ID"
INDEX	RANGE SCAN	IND_TICKET_SESSION	198	1	1			"T"."T_SESSION" = "S"."S_NUM"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME		F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS	✓	F_ID	UNIQUE	VALID	NORMAL	NO
	HALLS	PK_ROOMS	✓	R_NUM	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM3	✓	S_FILM,S_HALL,S_DATE	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS	✓	S_NUM	UNIQUE	VALID	NORMAL	NO
	TICKETS	IND_TICKET_SESSION	✓	T_SESSION	NONUNIQUE	VALID	NORMAL	NO
		PK_TICKETS		T_ID	UNIQUE	VALID	NORMAL	NO

Рис. 23. План выполнения запроса (I)

```
SELECT f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi') time, s.s_hall, count(*) cnt
FROM tickets t JOIN sessions s ON t.t_session=s.s_num JOIN films f ON s.s_film=f.f_id JOIN halls r ON r.r_num=s.s_hall
WHERE trunc(s.s_date)=trunc(sysdate) and s.s_date>sysdate AND t_status='free'
group by f.f_name, f.f_age, to_char(s.s_date, 'hh24:mi'), s.s_hall;
```

Results | Explain | Describe | Saved SQL | History

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *	Access Predicates
SELECT STATEMENT			6	1	8	294		
HASH	GROUP BY		6	1	8	294		
TABLE ACCESS	BY INDEX ROWID	TICKETS	148	1	5	1 480	"T""T_STATUS" = 'free'	
NESTED LOOPS			6	1	7	294		
NESTED LOOPS			1	1	2	39		
NESTED LOOPS			1	1	1	20		
INDEX	FULL SCAN	IND_SESSION_FILM4	1	1	1	17	"S"."S_DATE">SYSDATE@! AND TRUNC(INTERNAL_FUNCTION("S"."S_DATE")) = TRUNC(SYSDATE@!)	"S"."S_DATE">SYSDATE@!
INDEX	UNIQUE SCAN	PK_ROOMS	1	1	0	3		"S"."S_HALL" = "R"."R_NUM"
TABLE ACCESS	BY INDEX ROWID	FILMS	1	1	1	19		
INDEX	UNIQUE SCAN	PK_FILMS	1	1	0			"S"."S_FILM" = "F"."F_ID"
INDEX	RANGE SCAN	IND_TICKET_SESSION	198	1	1			"T"."T_SESSION" = "S"."S_NUM"

* Unindexed columns are shown in red

Index Columns

Owner	Table Name	Index Name	Used In Plan	Columns	Uniqueness	Status	Index Type	Join Index
LAB5	FILMS	IND_FILM_GENRE3		F_GENRE,F_RATING,F_COUNTRY,F_AGE	NONUNIQUE	VALID	NORMAL	NO
		IND_FILM_NAME		F_NAME	NONUNIQUE	VALID	NORMAL	NO
		PK_FILMS	✓	F_ID	UNIQUE	VALID	NORMAL	NO
	HALLS	PK_ROOMS	✓	R_NUM	UNIQUE	VALID	NORMAL	NO
	SESSIONS	IND_SESSION_FILM4	✓	S_FILM,S_HALL,S_DATE,S_NUM	NONUNIQUE	VALID	NORMAL	NO
		PK_SESSIONS		S_NUM	UNIQUE	VALID	NORMAL	NO
	TICKETS	IND_TICKET_SESSION	✓	T_SESSION	NONUNIQUE	VALID	NORMAL	NO
		PK_TICKETS		T_ID	UNIQUE	VALID	NORMAL	NO

Рис. 24. План выполнения запроса (I) после добавления в индекс поля **`s_num`**

Следовательно, окончательный список индексов будет таким:

```
create index ind_film_genre4 on films(f_genre,f_rating,f_country,f_age);
create index ind_film_name on films(f_name);
create index ind_session_film4 on sessions(s_film,s_hall,s_date,s_num);
create index ind_ticket_session on tickets(t_session);
```

2. ВЫПОЛНЕНИЕ ЛАБОРАТОРНОЙ РАБОТЫ

Выполнение лабораторной работы №5 заключается в анализе запросов из л.р. 2-3 и создании набора индексов, которые будут использоваться системой для выполнения этих запросов. Особенное внимание при этом стоит уделять запросам, которые чаще используются и/или содержат теоретико-множественные операции – соединение, пересечение, разность и объединение. После создания индексов необходимо убедиться в том, что система их использует при выполнении запросов: для этого надо посмотреть план выполнения запроса. В системе Oracle XE 11G и выше это можно сделать, перейдя после выполнения запроса на вкладку Explain (Рис. 3–Рис. 24). В системе Postgres в pgAdmin 4 это можно сделать, нажав после выполнения запроса на пиктограмму Explain Analyze (Рис. 25) и перейдя на вкладки Explain и Analyze.

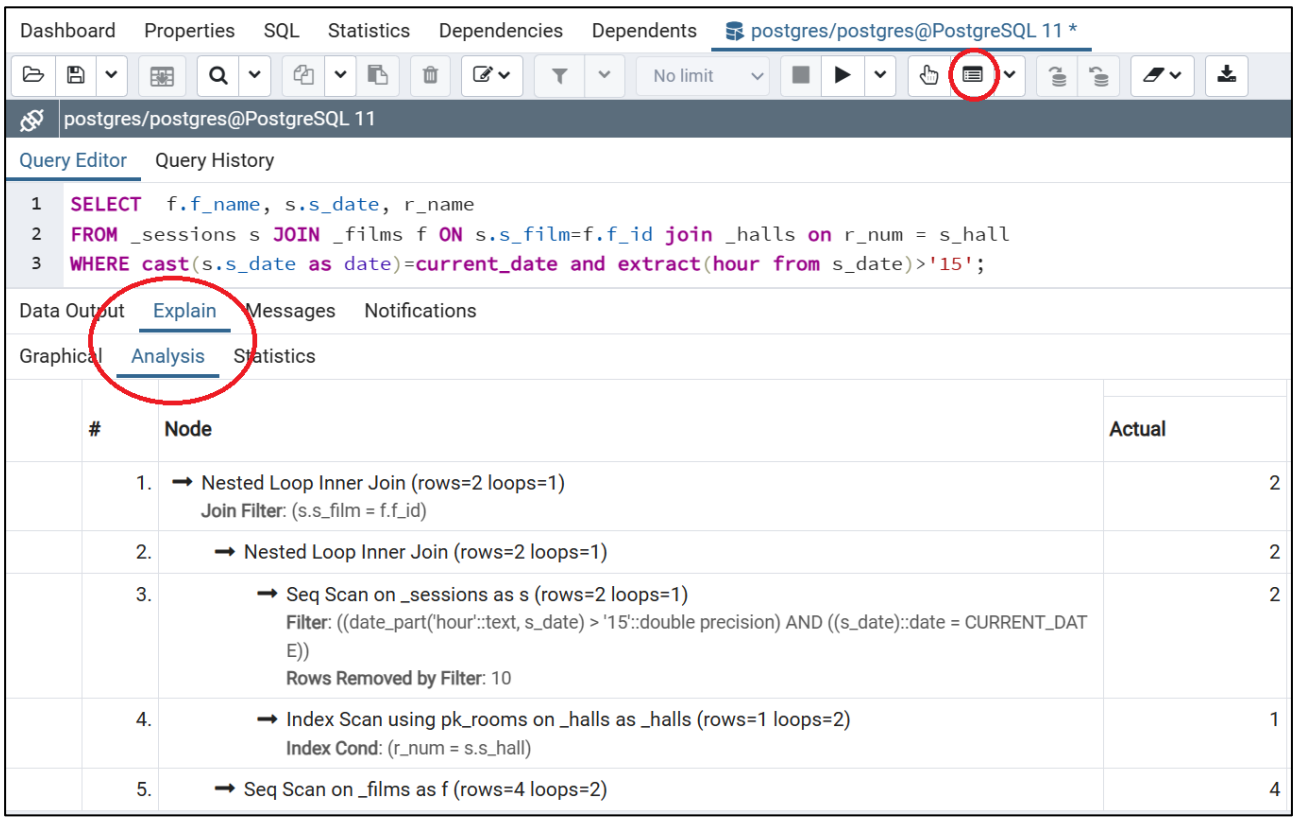


Рис. 25. Пример плана выполнения запроса в системе Postgres

Если система не пользуется созданным индексом, надо или удалить индекс, или изменить запрос так, чтобы система начала им пользоваться.

Защита лабораторной работы №5 заключается в демонстрации планов выполнения запросов с использованием индексов и обосновании того, почему по конкретным таблицам созданы именно такие индексы.

Библиографический список

1. Грабер М. Введение в SQL. – М.: Лори, 2008. – 378 с.
2. Oracle Database Concepts. Indexes and Index-Organized Tables. – https://docs.oracle.com/en/database/oracle/oracle-database/18/cncpt/indexes-and-index-organized-tables.html (Дата обращения 12.01.2024)
3. PostgreSQL: Documentation: Chapter 11. Indexes. Part II. The SQL Language. - https://www.postgresql.org/docs/current/indexes.html. (Дата обращения 12.01.2024)
4. Max Rokatansky. Погружение в индексы PostgreSQL. – https://habr.com/ru/companies/otus/articles/747882/ (Дата обращения 12.01.2024)
5. Chris Saxon. Как создавать и использовать индексы в БД Oracle. https://habr.com/ru/sandbox/165927/ (Дата обращения 12.01.2024).

Приложение 1.
Некоторые термины в плане запроса Oracle

План запроса имеет форму таблицы, один из столбцов которой описывает тип производимых сервером операций. Вот некоторые из них, которые встречаются наиболее часто:

TABLE ACCESS FULL – сервер последовательно прочитает все записи таблицы.

TABLE ACCESS BY INDEX ROWID – из таблицы будут прочитаны записи по значениям RowID, извлеченным из индекса.

INDEX RANGE SCAN – для получения выборки нужных записей будет использован индекс таблицы (сканирование в диапазоне значений).

INDEX UNIQIE SCAN – для получения выборки нужных записей будет использован индекс таблицы (чтение уникального значения).

HASH JOIN – для получения соединения таблиц будет построена хэш-таблица.

NESTED LOOPS – нужные записи соединения таблиц будут получены путем полного просмотра одной таблицы и поиском соответствующих записей во второй (возможно, с помощью индекса).

SORT MERGE JOIN – используется для соединения записей нескольких независимых источников. Сначала оба источника сортируются по соединяющему полю (первичному – внешнему ключу чаще всего), а затем происходит их слияние.

BUFFER SORT – в некоторых случаях Oracle может определить, что при выполнении запроса обращение к некоторому блоку данных может быть выполнено несколько раз, в этом случае Oracle помещает этот блок в специальную область, чтобы ускорить к нему доступ. Запрос может не иметь ключевого слова **SORT**, но при его выполнении будет вызвана эта операция.

MERGE JOIN CARTESIAN – для получения выборки нужных записей будет организовано декартово произведение записей в двух таблицах (для каждой записи основной таблицы будут просмотрены все записи вспомогательной). Это очень плохая операция, ее наличие в плане запроса говорит о том, что скорей всего упущена какая-то связка в JOIN.

Приложение 2.
Создание таблиц для фрагмента БД "Проектная организация"

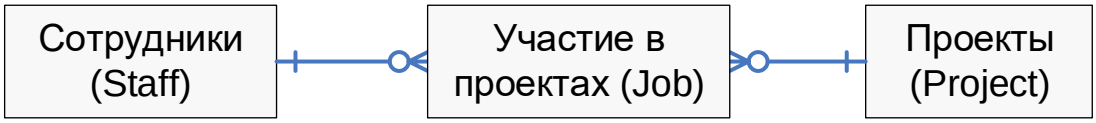


Рис. 2.А. Схема фрагмента базы данных "Проектная организация"

```
-- Сотрудники
create table staff(
id       numeric(6) primary key,      -- идентификатор сотрудника
name     varchar(60) not null,        -- ФИО
born     date not null ,              -- дата рождения
sex      char(2) not null,            -- пол
depno    numeric(4) not null,         -- номер отдела
post     varchar(30) not null,        -- должность
salary   numeric (8),                 -- оклад
passp    varchar(100) not null ,      -- паспортные данные
phone    varchar(20),                 -- телефон
addr     varchar(120)),               -- адрес
chief     numeric(6) references staff -- руководитель сотрудника
);
-- Проекты
create table project(
pid      numeric(4) primary key,
title    varchar(70) not null,        -- полное название проекта
client   varchar(50) not null,        -- заказчик
agreement varchar(30) not null,       -- сокращенное название
dbegin date not null,                 -- начало выполнения проекта
dend     date not null,               -- завершение выполнения проекта
cost     numeric(12) not null,        -- стоимость проекта
depno    numeric(4) not null          -- отдел, отвечающий за выполнение проекта
);
-- Участие сотрудников в проектах
create table job(
pid      numeric(4) not null  references project, -- номер проекта
id       numeric(6) not null  references staff,     -- идентификатор сотрудника
rel      varchar(30) not null,        -- роль участника (исполнитель, руководитель, консультант)
dbegin date not null,                 -- начало участия в проекте
dend     date                         -- окончание участия в проекте
);
```

Приложение 3.

Создание и заполнение таблиц для БД "Кинотеатр"

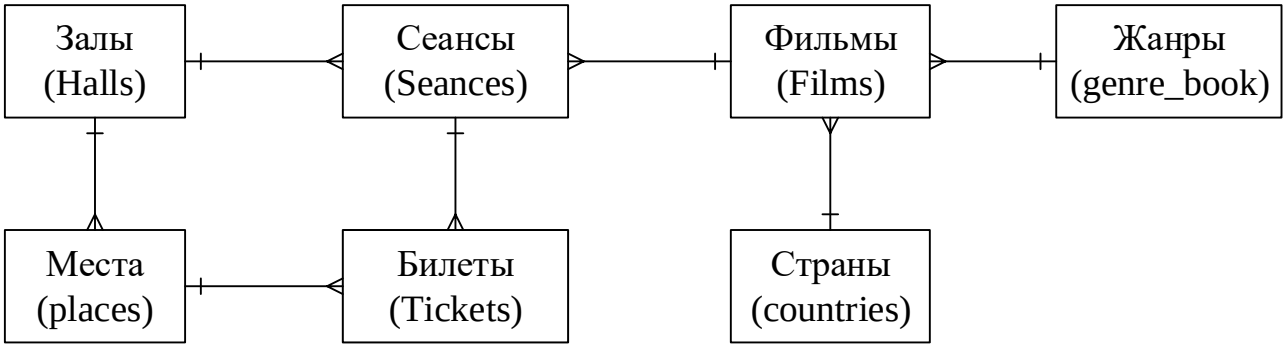


Рис. **3.А**. Схема БД кинотеатра

-- Создание таблиц для Oracle

```
CREATE TABLE genre_book (
genre   VARCHAR(15) CONSTRAINT pk_genre_book PRIMARY KEY);

CREATE TABLE countries (
code      CHAR(3) CONSTRAINT pk_countries PRIMARY KEY,
country VARCHAR(30) NOT NULL);

CREATE TABLE films (
f_id NUMERIC(5) CONSTRAINT pk_films PRIMARY KEY,
f_name     VARCHAR(80) NOT NULL,
f_genre     VARCHAR(15) CONSTRAINT fk_genre REFERENCES genre_book,
f_director VARCHAR(80),
f_studio    VARCHAR(20) NOT NULL,
f_rating     NUMERIC(3,1) CONSTRAINT ch_rating CHECK(f_rating >= 0 AND f_rating <= 10),
f_year       NUMERIC(4) NOT NULL,
f_age         NUMERIC(2) NOT NULL CONSTRAINT ch_age CHECK(f_age IN (0,6,12,16,18)),
f_duration NUMERIC(3) NOT NULL check (f_duration > 10),
f_country  CHAR(3) NOT NULL CONSTRAINT fk_f_country REFERENCES countries);

CREATE TABLE halls (
r_num      NUMERIC(1) CONSTRAINT pk_rooms PRIMARY KEY,
r_rows     NUMERIC(2) NOT NULL CONSTRAINT ch_ROW CHECK(r_rows > 0),
r_places   NUMERIC(2) NOT NULL CONSTRAINT ch_PLACE CHECK(r_places > 0),
r_name    VARCHAR(20) NOT NULL);

CREATE TABLE places (
hall       NUMERIC(1) CONSTRAINT fk_s_rooms REFERENCES halls,
nrow      NUMERIC(2) CONSTRAINT check_row CHECK(nrow > 0),
nplace    NUMERIC(2) CONSTRAINT check_place CHECK(nplace > 0),
PRIMARY KEY(hall, nrow, nplace));

CREATE TABLE sessions (
s_num   NUMERIC(6) CONSTRAINT pk_sessions PRIMARY KEY,
s_date    DATE NOT NULL,
```

```
s_hall    NUMERIC(1) NOT NULL CONSTRAINT fk_s_hall REFERENCES halls,
s_film    NUMERIC(5) NOT NULL CONSTRAINT fk_s_film REFERENCES films);

CREATE TABLE tickets (
t_id          NUMERIC(10) CONSTRAINT pk_tickets PRIMARY KEY,
t_hall        NUMERIC(1) NOT NULL,
t_row         NUMERIC(2) NOT NULL,
t_place       NUMERIC(2) NOT NULL,
t_status      CHAR(6) default 'free' NOT NULL,
t_date        DATE,
t_client      VARCHAR(20),
t_session   NUMERIC(6) NOT NULL CONSTRAINT fk_t_session REFERENCES sessions,
CONSTRAINT check_status check(t_status in ('free', 'booked', 'sold')),
CONSTRAINT fk_comb foreign key(t_hall, t_row, t_place) REFERENCES places);
```

-- Заполнение таблиц для Oracle

```
declare n number:=1;
begin
  delete from tickets;
  delete from sessions;
  delete from places;
  delete from halls;
  delete from films;
  delete from genre_book;
  delete from countries;

  INSERT INTO genre_book VALUES('comedy');
  INSERT INTO genre_book VALUES('drama');
  INSERT INTO genre_book VALUES('melodrama');
  INSERT INTO genre_book VALUES('fantastic');
  INSERT INTO genre_book VALUES('detective');
  INSERT INTO genre_book VALUES('thriller');

  INSERT INTO countries VALUES('RUS', 'Russia');
  INSERT INTO countries VALUES('GBR', 'Great Britain');
  INSERT INTO countries VALUES('ESP', 'Spain');
  INSERT INTO countries VALUES('USA', 'USA');
  INSERT INTO countries VALUES('FRA', 'France');

  insert into films values(15, 'La bete', 'fantastic', 'Bertrand Bonello', 'IMDbPro', 6.6, 2023, '16', 145,
'FRA');
  insert into films values(1, 'Pulp fiction', 'thriller', 'Quentin Tarantino', 'Paramount', 7.7, 1999, '12',
128, 'USA');
  insert into films values(2, 'One in Hollywood', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2020, '12', 144, 'USA');
  insert into films values(3, 'Django unchained', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2012, '12', 170, 'USA');
  insert into films values(4, 'La La land', 'comedy', 'Damien of Shazell', 'ND Play', 7.8, 2016, '6',
128, 'FRA');
  insert into films values(5, 'Once in Hollywood', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2020, '16', 156, 'USA');
```

```
 insert into films values(10, 'Sherlock Holmes', 'detective', 'Guy Ritchie', 'Universal', 8.7, 2011, '16', 107, 'USA');
 insert into films values(11, 'Gorod Zero', 'fantastic', 'Karen Shakhnazarov', 'Mosfilm', 9.0, 1988, '16', 95, 'RUS');
 insert into films values(12, 'Kray', 'drama', 'Alexei Uchitel', 'Mosfilm', 8.7, 2010, '16', 99, 'RUS');
 insert into films values(13, 'Matilda', 'drama', 'Alexei Uchitel', 'Mosfilm', 7.7, 2017, '16', 73, 'RUS');
 insert into films values(14, 'Kill Bill', 'detective', 'Quentin Tarantino', 'Paramount', 9.0, 1999, '18', 155, 'USA');

 INSERT INTO halls VALUES(1, 15, 20, 'Fellili');
 INSERT INTO halls VALUES(2, 18, 15, 'Bergman');
 INSERT INTO halls VALUES(3, 10, 15, 'Romm');
 INSERT INTO halls VALUES(4, 6, 5, 'Disney');
 INSERT INTO halls VALUES(5, 5, 8, 'Kurosawa');

 for R in (select * from halls) loop
  for i in 1..R.r_rows loop
   for j in 1..R.r_places loop
    insert into places values(R.r_num, i, j);
   end loop;
  end loop;
 end loop;
 commit;

 insert into sessions values(1, to_date(to_char(sysdate, 'dd.mm.yyyy')||' 12:00:00', 'dd.mm.yyyy hh24:mi:ss'), 2, 1);
 insert into sessions values(2, to_date(to_char(sysdate, 'dd.mm.yyyy')||' 16:00:00', 'dd.mm.yyyy hh24:mi:ss'), 2, 2);
 insert into sessions values(3, to_date(to_char(sysdate, 'dd.mm.yyyy')||' 19:00:00', 'dd.mm.yyyy hh24:mi:ss'), 2, 3);
 insert into sessions values(4, to_date(to_char(sysdate+1, 'dd.mm.yyyy')||' 14:00:00', 'dd.mm.yyyy hh24:mi:ss'), 1, 4);
 insert into sessions values(5, to_date(to_char(sysdate+1, 'dd.mm.yyyy')||' 17:00:00', 'dd.mm.yyyy hh24:mi:ss'), 1, 4);
 insert into sessions values(6, to_date(to_char(sysdate+1, 'dd.mm.yyyy')||' 20:00:00', 'dd.mm.yyyy hh24:mi:ss'), 1, 5);
 insert into sessions values(7, to_date(to_char(sysdate-1, 'dd.mm.yyyy')||' 14:00:00', 'dd.mm.yyyy hh24:mi:ss'), 3, 12);
 insert into sessions values(8, to_date(to_char(sysdate-1, 'dd.mm.yyyy')||' 17:30:00', 'dd.mm.yyyy hh24:mi:ss'), 3, 12);
 insert into sessions values(9, to_date(to_char(sysdate-1, 'dd.mm.yyyy')||' 20:30:00', 'dd.mm.yyyy hh24:mi:ss'), 3, 11);
 insert into sessions values(10,to_date(to_char(sysdate+2, 'dd.mm.yyyy')||' 12:00:00', 'dd.mm.yyyy hh24:mi:ss'), 5, 14);
 insert into sessions values(11,to_date(to_char(sysdate+2, 'dd.mm.yyyy')||' 14:00:00', 'dd.mm.yyyy hh24:mi:ss'), 5, 13);
 insert into sessions values(12,to_date(to_char(sysdate+2, 'dd.mm.yyyy')||' 20:00:00', 'dd.mm.yyyy hh24:mi:ss'), 5, 14);

 for R in (select * from sessions s, places p where s_hall=hall) loop
  insert into tickets values(n, R.hall, R.nrow, R.nplace, 'free', null, null, R.s_num);
```

```
   n := n+1;
  end loop;
  update tickets set t_status = 'sold' where t_hall*(t_row+t_place)< 25;
  update tickets set t_status = 'sold' where t_session = 11;
  commit;
end;
/
```

-- Создание таблиц для Postgres

```
drop TABLE if exists _tickets;
drop TABLE if exists _sessions;
drop TABLE if exists _places;
drop table if exists _halls;
drop TABLE if exists _films ;

CREATE TABLE if not exists _genre_book (
genre VARCHAR(15) CONSTRAINT pk_genre_book PRIMARY KEY);

CREATE TABLE if not exists _countries (
code CHAR(3) CONSTRAINT pk_countries PRIMARY KEY,
country VARCHAR(30) NOT NULL);

CREATE TABLE if not exists _films (
f_id NUMERIC(5) CONSTRAINT pk__films PRIMARY KEY,
f_name     VARCHAR(80) NOT NULL,
f_genre    VARCHAR(15) CONSTRAINT fk_genre REFERENCES _genre_book,
f_director VARCHAR(80),
f_studio   VARCHAR(20) NOT NULL,
f_rating   NUMERIC(3,1) CONSTRAINT ch_rating CHECK(f_rating >= 0 AND f_rating <= 10),
f_year     NUMERIC(4) NOT NULL,
f_age      NUMERIC(2) NOT NULL CONSTRAINT ch_age CHECK(f_age IN (0,6,12,16,18)),
f_duration NUMERIC(3) NOT NULL check (f_duration > 10),
f_country  CHAR(3) NOT NULL CONSTRAINT fk_f_country REFERENCES _countries);

CREATE TABLE if not exists _halls (
r_num NUMERIC(1) CONSTRAINT pk_rooms PRIMARY KEY,
r_rows NUMERIC(2) NOT NULL CONSTRAINT ch_ROW CHECK(r_rows > 0),
r_places NUMERIC(2) NOT NULL CONSTRAINT ch_PLACE CHECK(r_places > 0),
r_name     VARCHAR(20) NOT NULL);

CREATE TABLE if not exists places (
hall NUMERIC(1) CONSTRAINT fk_s_rooms REFERENCES _halls,
nrow NUMERIC(2) CONSTRAINT check_row CHECK(nrow > 0),
nplace NUMERIC(2) CONSTRAINT check_place CHECK(nplace > 0),
PRIMARY KEY(hall, nrow, nplace));

CREATE TABLE if not exists _sessions (
s_num NUMERIC(6) CONSTRAINT pk__sessions PRIMARY KEY,
s_date TIMESTAMP NOT NULL,
s_hall NUMERIC(1) NOT NULL CONSTRAINT fk_s_hall REFERENCES _halls,
s_film NUMERIC(5) NOT NULL CONSTRAINT fk_s_film REFERENCES _films);
```

```
CREATE TABLE if not exists _tickets (
t_id NUMERIC(10) CONSTRAINT pk__tickets PRIMARY KEY,
t_hall  NUMERIC(1) NOT NULL,
t_row   NUMERIC(2) NOT NULL,
t_place NUMERIC(2) NOT NULL,
t_status CHAR(6) default 'free' NOT NULL,
t_date   DATE,
t_client VARCHAR(20),
t_session NUMERIC(6) NOT NULL CONSTRAINT fk_t_session REFERENCES _sessions,
CONSTRAINT check_status check(t_status in ('free', 'booked', 'sold')),
CONSTRAINT fk_comb foreign key(t_hall, t_row, t_place) REFERENCES places);
```

-- Заполнение таблиц для Postgres

```
INSERT INTO _genre_book VALUES('comedy');
INSERT INTO _genre_book VALUES('drama');
INSERT INTO _genre_book VALUES('melodrama');
INSERT INTO _genre_book VALUES('fantastic');
INSERT INTO _genre_book VALUES('detective');
INSERT INTO _genre_book VALUES('thriller');

INSERT INTO _countries VALUES('RUS', 'Russia');
INSERT INTO _countries VALUES('GBR', 'Great Britain');
INSERT INTO _countries VALUES('ESP', 'Spain');
INSERT INTO _countries VALUES('USA', 'USA');
INSERT INTO _countries VALUES('FRA', 'France');

insert into _films values(15, 'La bete', 'fantastic', 'Bertrand Bonello', 'IMDbPro', 6.6, 2023, '16', 145,
'FRA');
insert into _films values(1, 'Pulp fiction', 'thriller', 'Quentin Tarantino', 'Paramount', 7.7, 1999, '12',
128, 'USA');
insert into _films values(2, 'One in Hollywood', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2020, '12', 144, 'USA');
insert into _films values(3, 'Django unchained', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2012, '12', 170, 'USA');
insert into _films values(4, 'La La land', 'comedy', 'Damien of Shazell', 'ND Play', 7.8, 2016, '6',
128, 'FRA');
insert into _films values(5, 'Once in Hollywood', 'comedy', 'Quentin Tarantino', 'Paramount', 8.7,
2020, '16', 156, 'USA');
insert into _films values(10, 'Sherlock Holmes', 'detective', 'Guy Ritchie', 'Universal', 8.7, 2011, '16',
107, 'USA');
insert into _films values(11, 'Gorod Zero', 'fantastic', 'Karen Shakhnazarov', 'Mosfilm', 9.0, 1988,
'16', 95, 'RUS');
insert into _films values(12, 'Kray', 'drama', 'Alexei Uchitel', 'Mosfilm', 8.7, 2010, '16', 99, 'RUS');
insert into _films values(13, 'Matilda', 'drama', 'Alexei Uchitel', 'Mosfilm', 7.7, 2017, '16', 73,
'RUS');
insert into _films values(14, 'Kill Bill', 'detective', 'Quentin Tarantino', 'Paramount', 9.0, 1999, '18',
155, 'USA');

INSERT INTO _halls VALUES(1, 15, 20, 'Fellili');
INSERT INTO _halls VALUES(2, 18, 15, 'Bergman');
```

```
INSERT INTO _halls VALUES(3, 10, 15, 'Romm');
INSERT INTO _halls VALUES(4, 6, 5, 'Disney');
INSERT INTO _halls VALUES(5, 5, 8, 'Kurosawa');

CREATE  or replace  procedure  insert_places() AS $$
DECLARE    R  RECORD;
begin
  for R in (select * from _halls) loop
    for i in 1..R.r_rows loop
      for j in 1..R.r_places loop
        insert into places values(R.r_num, i, j);
      end loop;
    end loop;
  end loop;
  commit;
end;
$$ LANGUAGE plpgsql;

call insert_places();

insert into _sessions values(1, (CAST(current_date AS varchar)||' 12:00:00')::timestamp, 2, 1);
insert into _sessions values(2, (CAST(current_date AS varchar)||' 16:00:00')::timestamp, 2, 2);
insert into _sessions values(3, (CAST(current_date AS varchar)||' 19:00:00')::timestamp, 2, 3);
insert into _sessions values(4, (CAST(current_date+1 AS varchar)||' 14:00:00')::timestamp, 1, 4);
insert into _sessions values(5, (CAST(current_date+1 AS varchar)||' 17:00:00')::timestamp, 1, 4);
insert into _sessions values(6, (CAST(current_date+1 AS varchar)||' 20:00:00')::timestamp, 1, 5);
insert into _sessions values(7, (CAST(current_date+1 AS varchar)||' 14:00:00')::timestamp, 3, 12);
insert into _sessions values(8, (CAST(current_date+1 AS varchar)||' 17:30:00')::timestamp, 3, 12);
insert into _sessions values(9, (CAST(current_date+1 AS varchar)||' 20:30:00')::timestamp, 3, 11);
insert into _sessions values(10,(CAST(current_date+1 AS varchar)||' 12:00:00')::timestamp, 5, 14);
insert into _sessions values(11,(CAST(current_date+1 AS varchar)||' 14:00:00')::timestamp, 5, 13);
insert into _sessions values(12,(CAST(current_date+1 AS varchar)||' 20:00:00')::timestamp, 5, 14);

CREATE  or replace  procedure  insert_tickets() AS $$
declare n numeric:=1;
R record;
begin
  for R in (select * from _sessions s, places p where s_hall=hall) loop
    insert into _tickets values(n, R.hall, R.nrow, R.nplace, 'free', null, null, R.s_num);
    n := n+1;
  end loop;
  commit;
end;
$$ LANGUAGE plpgsql;

call insert_tickets();

update _tickets set t_status = 'sold' where t_hall*(t_row+t_place)< 20;
```