Хранимые процедуры пишутся на специальном встроенном языке программирования, они могут включать любые операторы SQL, а также включают некоторый набор операторов, управляющих ходом выполнения программ, которые во многом схожи с подобными операторами процедурно ориентированных языков программирования. В коммерческих СУБД для написания текстов хранимых процедур используются собственные языки программирования, так, в СУБД Oracle для этого используется язык PL /SQL, а в MS SQL Server и Systemll фирмы Sybase используется язык Transact SQL. В последних версиях Oracle объявлено использование языка Java для написания хранимых процедур.

Хранимые процедуры являются объектами БД. Каждая хранимая процедура компилируется при первом выполнении, в процессе компиляции строится оптимальный план выполнения процедуры. Описание процедуры совместно с планом ее выполнения хранится в системных таблицах БД.

Для создания хранимой процедуры применяется оператор SQL CREATE PROCEDURE.

По умолчанию выполнить хранимую процедуру может только ее владелец, которым является владелец БД, и создатель хранимой процедуры. Однако владелец хранимой процедуры может делегировать права на ее запуск другим пользователям. В MS SQL Server хранимая процедура создается оператором:

CREATE PROCEDURE] <имя_процедуры> [;<версия>]

[{@параметр1 тип_данных}

[VARYING] [= <значение_по_умолчанию>] [OUTPUT]]

[..параметрN..]

[WITH

{ RECOMPILE

| ENCRYPTION

| RECOMPILE, ENCRYPTION}] [FOR REPLICATION]

AS *Тело процедуры*

Необязательное ключевое слово VARYING определяет заданное значение по умолчанию для определенного ранее параметра.

Ключевое слово RECOMPILE определяет режим компиляции создаваемой хранимой процедуры. Если задано ключевое слово RECOMPILE, то процедура будет перекомпилиро-ваться каждый раз, когда она будет вызываться на исполнение. Это может резко замедлить исполнение процедуры, но предыдущий план исполнения, составлен-ный при первом вызове, может быть абсолютно неэффективен при последующих вызовах. Ключевое слово ENCRYPTION определяет режим, при котором исходный текст хранимой процедуры не сохраняется в БД. Такой режим применяется для того, чтобы сохранить авторское право на интеллектуальную продукцию, которой и являются хранимые процедуры, но в случае восстановления БД после серьезной аварии для перекомпиляции потребуются первоначальные исходные тексты всех хранимых процедур .

Однако кроме имени хранимой процедуры все остальные параметры являются необязательными. Процедуры могут быть процедурами или процедурами-функциями. И эти понятия здесь трактуются традиционно, как в языках программирования высокого уровня. Процедура в явном виде не возвращает значение, но в ней может быть использовано ключевое слово OUTPUT, которое определяет, что данный параметр является выходным

Пример./* процедура проверки наличия экземпляров данной книги

параметры:

@ISBN шифр книги

процедура возвращает параметр, равный количеству экземпляров. Если возвращается ноль, то это значит, что нет свободных экземпляров данной книги в библиотеке

*/

```
CREATE PROCEDURE COUNT_EX (@ISBN varchar(12)) AS
```

```
/* определим внутреннюю переменную */
DECLARE @TEK_COUNT int
/* выполним соответствующий оператор
SELECT
Будем считать только экземпляры, которые в настоящий
    момент находятся не на руках у читателей, а в библиотеке
    */
select @TEK_COUNT = select count(*)
FROM EXEMPLAR WHERE ISBN = @ISBN
AND READERJD Is NULL AND EXIST = True
/* 0 - ноль означает, что нет ни одного свободного экземпляра
    данной книги в библиотеке */
RETURN @TEK_COUNT
```

Хранимая процедура может быть вызвана несколькими способами. Простейший способ — это использование оператора:

EXEC <имя мроцедуры> <значение входного_параметра1>...
<имя_переменной_для_выходного параметра1>...

При этом все входные и выходные параметры должны быть заданы обязательно и в том порядке, в котором они определены в процедуре. Если определено несколько версий хранимой процедуры, то при вызове можно указать номер конкретной версии для исполнения. Например, в версии 2 процедуры COUNT_EX последний оператор исполнения этой процедуры имеет вид:

EXEC @Ntek = COUNT_EX:2 @ISBN

Однако если в процедуре значения параметров определены по умолчанию, то при запуске процедуры могут быть указаны значения не всех параметров. В этом случае оператор вызова процедуры может быть записан в следующем виде:

EXEC <имя процедуры> <имя_параметра1>=<значение параметра1>...
<имя_параметраN>=<значение параметраN>..

Если мы задаем параметры по именам, то необязательно задавать их в том порядке, в котором они описаны при создании процедуры.

Каждая хранимая процедура является объектом БД. Она имеет уникальное имя и уникальный внутренний номер в системном каталоге. При изменении текста хранимой процедуры мы должны сначала уничтожить данную процедуру как объект, хранимый в БД, и только после этого записать на ее место новую. Следует отметить, что при удалении хранимой процедуры удаляются одновременно все ее версии, нельзя удалить только одну версию хранимой процедуры. Для того чтобы автоматизировать процесс уничтожения старой процедуры и замены ее на новую, в начале текста хранимой процедуры можно выполнить проверку наличия объекта типа «хранимая процедура» с данным именем в системном каталоге и при наличии описания данного объекта удалить его из системного каталога

```
/* проверка существования в системном каталоге объекта с данным
   именем и типом, созданного владельцем БД */
If exists (select * from sysobjects where id =
   object_id('dbo.NEW_BOOKS') and sysstat & Oxf = 4)
/* если объект существует, то сначала его удалим из системного
   каталога */
drop procedure dbo.NEW_BOOKS
GO
CREATE PROCEDURE NEW_BOOKS (@ISBN varchar(12).@TITL
   varchar(255),@AUTOR
varcharOO),@COAUTOR varchar(30) @YEARIZD int,@PAGES
   INT.@NUM_EXEMPL INT)
```

```
/* процедура ввода новой книги с указанием количества
   экземпляров данной книги параметры
@ISBN varchar(12) шифр книги
@TITL varchar(255) название
@AUTOR varchar(30) автор
@COAUTOR varchar(30) соавтор
@YEARIZD Int год издания
@PAGES INT количество страниц
@NUM_EXEMPL INT количество экземпляров
*/
AS
/*опишем переменную, в которой будет храниться количество
   оставшихся не оприходованных экземпляров книги, т.е. таких,
   которым еще не заданы инвентарные номера */
DECLARE @TEK Int
```

```
/ *вводим данные о книге в таблицу BOOKS */
INSERT INTO BOOKS VALUES(@ISBN@TITL
    @AUTOR@COAUTOR.@YEARIZD.@PAGES)
/* назначение значения текущего счетчика осташихся к вводу
    экземпляров*/
SELECT @ТЕК = @NUM_EXEMPL
/* организуем цикл для ввода новых экземпляров данной книги */
WHILE @TEK>0 /* пока количество оставшихся экземпляров
    больше нуля */
BEGIN
/* так как для инвентарного номера экземпляра книги мы задали
    свойство IDENTITY, то нам не надо вводить инвентарный номер.
    СУБД сама автоматически вычислит его, добавив единицу к
    предыдущему, введет при выполнении оператора ввода INSERT.
```

Поле, определяющее присутствие экземпляра в библиотеке (EXIST) - логическое поле, мы введем туда значение TRUE.которое соответствует присутствию экземпляра книги в библиотеке. Если даты взятия и возврата не указаны, тогда по умолчанию СУБД подставит туда значение, соответствующее 1 января 2000 года, если мы не хотим хранить такие бессмысленные данные, то можем ввести для обоих полей дата время, значения текущей даты. */

SELECT @INV = SELECT MAX(ID_EXEMPLAR) FROM EXEMPLAR

/* организуем цикл для ввода новых экземпляров данной книги */

WHILE @ТЕК>0 /* пока количество оставшихся экземпляров больше нуля */

BEGIN

insert into EXEMPLAR (ID_EXEMPLAR ISBN.DATA_IN.DATA_OUT,EXIST),

VALUES (@INV,@ISBN.GETDATE(),GetDate(). TRUE)

```
/* изменение текущих значений счетчика и инвентарного номера */
SELECT @ТЕК = @ТЕК - 1
SELECT @INV = @INV + 1
End /* конец цикла ввода данных о экземпляре книги*/ GO
```

Хранимые процедуры могут вызывать одна другую. Создадим хранимую процедуру, которая возвращает номер читательского билета для конкретного читателя.

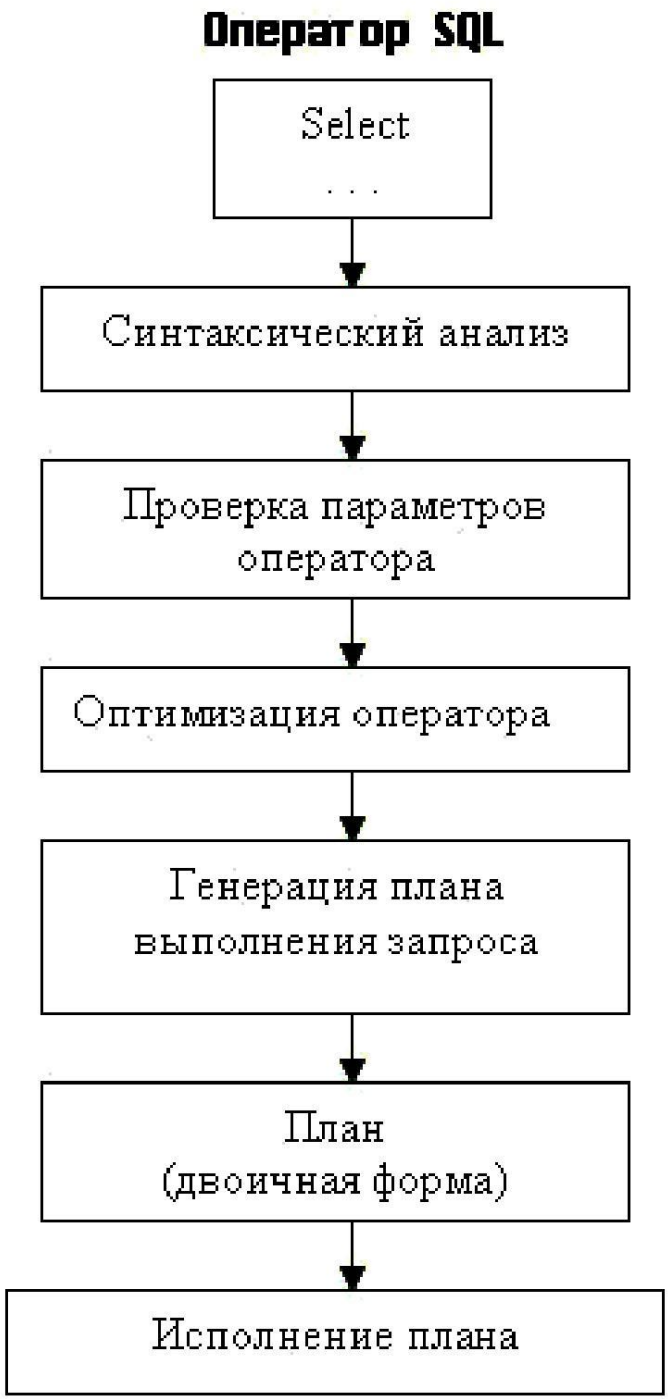
```
if exists (select * from sysobjects
where id = object_id('dbo. CK_READER') and sysstat & Oxf = 4)
/* если объект существует, то сначала его удалим из системного
   каталога */ drop procedure dbo.CK_READER
/* Процедура возвращает номер читательского билета, если
   читатель есть и 0 в противном случае. В качестве параметров
   передаем фамилию и дату рождения */
CREATE PROCEDURE CK_READER (@FIRST_NAME varchar(30)
   .(PBIRTH_DAY varchar(12))
AS
```

Хранимые процедуры допускают наличие нескольких выходных параметров. Для этого каждый выходной параметр должен после задания своего типа данных иметь дополнительное ключевое слово OUTPUT.

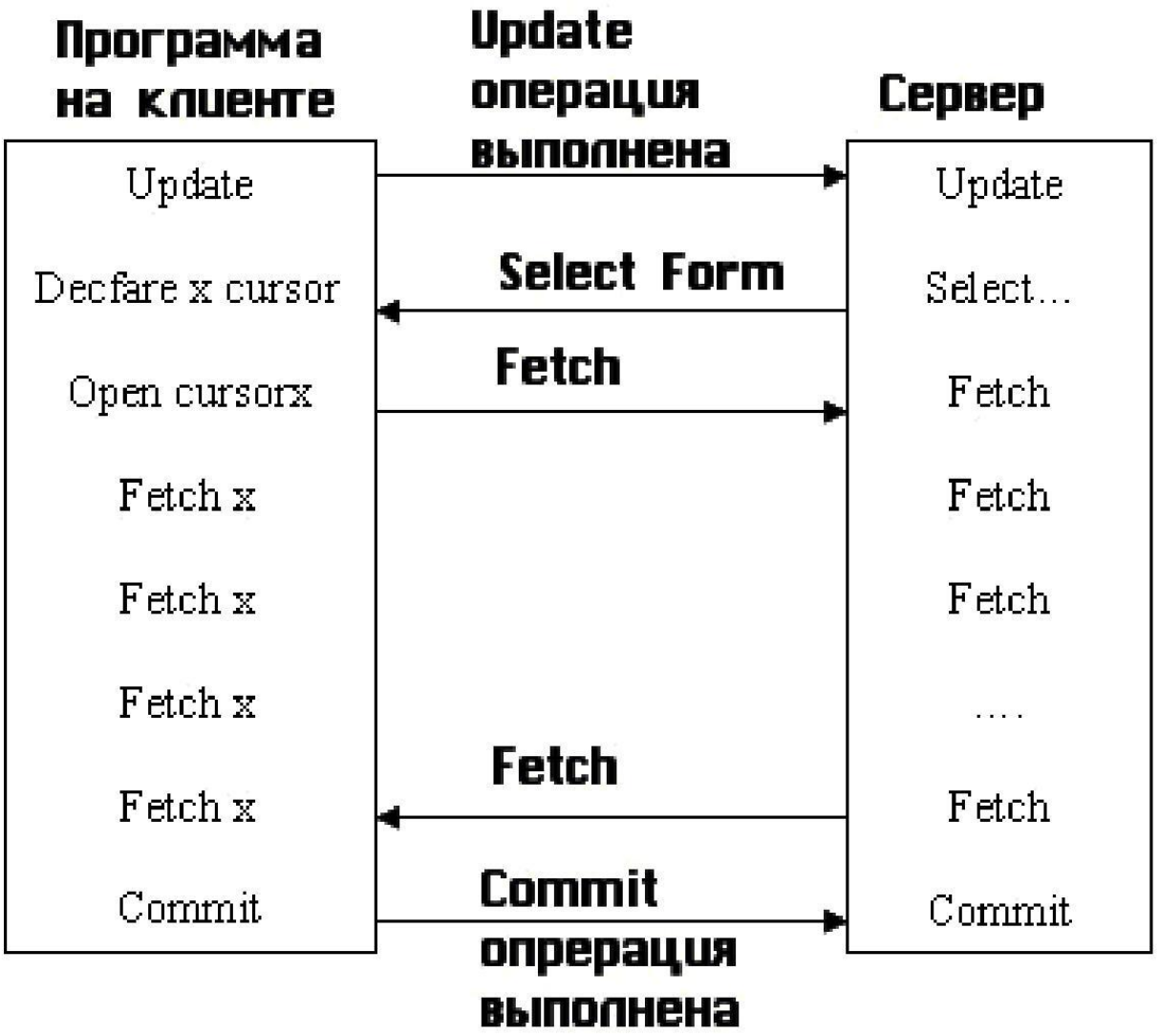
Теперь обратимся к оценке эффективности применения хранимых процедур.

Если рассмотреть этапы выполнения одинакового текста части приложения, содержащего SQL-операторы, самостоятельно на клиенте и в качестве хранимой процедуры, то можно отметить, что на клиенте выполняются все 5 этапов выполнения SQL-операторов, а хранимая процедура может храниться в БД в уже скомпилированном виде, и ее исполнение займет гораздо меньше времени. Кроме того, хранимые процедуры, как уже упоминалось, могут быть использованы несколькими приложениями, а встроенные операторы SQL должны быть включены в каждое приложение повторно.

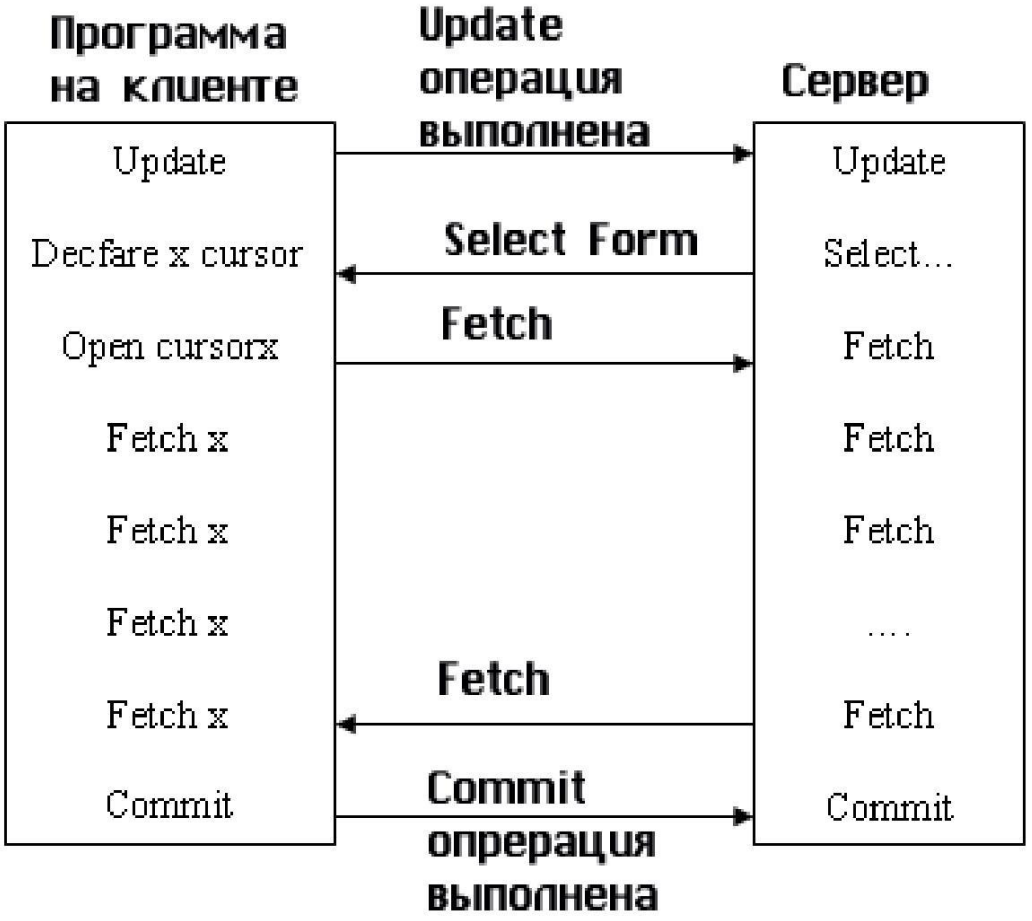
Процесс выполнения операторов SQL на клиенте и процесс выполнения хранимой процедуры



Хранимые процедуры также играют ключевую роль в повышении быстродействия при работе в сети с архитектурой «клиент—сервер». Пример выполнения последовательности операторов SQL на клиенте



Пример выполнения той же последовательности операторов SQL, оформленных в виде хранимой процедуры. В этом случае клиент обращается к серверу только для выполнения команды запуска хранимой процедуры. Сама хранимая процедура выполняется на сервере. Объем пересылаемой по сети информации резко сокращается во втором случае.



Триггеры

Фактически триггер — это специальный вид хранимой процедуры, которую **SQL** Server вызывает при выполнении операций модификации соответствующих таблиц. Триггер автоматически активизируется при выполнении операции, с которой он связан. Триггеры связываются с одной или несколькими операциями модификации над одной таблицей.

В разных коммерческих СУБД рассматриваются разные триггеры. Так, в MS SQL Server триггеры определены только как постфильтры, то есть такие триггеры, которые выполняются после свершения события. В СУБД Oracle определены два типа триггеров: триггеры, которые могут быть запущены перед реализацией операции модификации, они называются BEFORE-триггерами, и триггеры, которые активизируются после выполнения соответствующей модификации, аналогично триггерам MS SQL Server, — они называются AFTER-тригерами.

Триггеры могут быть эффективно использованы для поддержки семантической целостности БД, однако приоритет их ниже, чем приоритет правил-ограничений (constraints), задаваемых на уровне описания таблиц и на уровне связей между таблицами. При написании триггеров всегда надо помнить об этом, при нарушении правил целостности по связям (DRI declarative Referential Integrity) триггер просто может никогда не сработать.

В стандарте SQL1 ни хранимые процедуры, ни триггеры были не определены. Но в добавлении к стандарту SQL2, выпущенному в 1996 году, те и другие объекты были стандартизированы и определены. Для создания триггеров используется специальная команда:

CREATE TRIGGER <имя_триггера> ON <имя_таблицы>

FOR {[INSERT][. UPDATE] [, DELETE] } [WITH ENCRIPTING]

AS

SQL-операторы (Тело триггера)

Имя триггера является идентификатором во встроенном языке программирования СУБД и должно удовлетворять соответствующим требованиям.
В параметре FOR задается одна или несколько операций модификации, которые запускают данный триггер.
Параметр WITH ENCRIPTING имеет тот же смысл, что и для хранимых процедур, он скрывает исходный текст тела триггера.
Существует несколько правил, которые ограничивают набор операторов, которые могут быть использованы в теле триггера.

В большинстве СУБД действуют следующие ограничения:
-Нельзя использовать в теле триггера операции создания объектов БД (новой БД, новой таблицы, нового индекса, новой хранимой процедуры, нового триггера, новых индексов, новых представлений).
-Нельзя использовать в триггере команду удаления объектов DROP для всех типов базовых объектов БД.

-Нельзя использовать в теле триггера команды изменения базовых объектов ALTER TABLE, ALTER DATABASE.
-Нельзя изменять права доступа к объектам БД, то есть выполнять команду GRAND или REVOKE.
-Нельзя создать триггер для представления (VIEW) .
-В отличие от хранимых процедур, триггер не может возвращать никаких значений, он запускается автоматически сервером и не может связаться самостоятельно ни с одним клиентом.
Рассмотрим пример триггера, который срабатывает при удалении экземпляра некоторой книги, например, в случае утери этой книги читателем.
А он может выполнять следующую проверку: проверять, остался ли еще хоть один экземпляр данной книги в библиотеке, и если это был последний экземпляр книги в библиотеке, то резонно удалить описание книги из предметного каталога, чтобы наши читатели зря не пытались заказать эту книгу.

```
/*Проверка существования данного триггера в системном каталоге */
if exists (select * from sysobjects
where id = object_id('dbo.DEL_EXEMP') and sysstat & Oxf = 8)
drop trigger dbo.DEL_EXEMP
GO
CREATE TRIGGER DEL_EXEMP ON dbo.EXEMPLAR
/* мы создаем триггер для таблицы EXEMPLAR */
FOR DELETE /* только для операции удаления */
AS
/* опишем локальные переменные */
DECLARE @Ntek int
/* количество оставшихся экземпляров удаленной книги */
DECLARE @DEL_EX VARCHAR(12)
/* шифр удаленного экземпляра*/
Begin
```

```
/* по временной системной таблице, содержащей удаленные записи,
   определяем шифр книги, соответствующей последнему
   удаленному экземпляру */
SELECT @DEL_EX = ISBN From deleted
/* вызовем хранимую процедуру, которая определит количество
   экземпляров книги с заданным шифром */
EXEC @Ntek = COUNT_EX @DEL_EX
/* Если больше нет экземпляров данной книги, то мы удаляем
   запись о книге из таблицы BOOKS */
IF @Ntek = 0 DELETE from BOOKS WHERE BOOKS.ISBN =
   @DEL_EXENDGO
```

Динамический SQL

В статическом SQL вся информация об операторе SQL известна на момент компиляции. Однако очень часто в диалоговых программах требуется более гибкая форма выполнения операторов SQL. Текст оператора SQL формируется уже во время выполнения программы.

Если мы снова вернемся к этапам выполнения SQL-операторов, то первые четыре действия, связанные с синтаксическим анализом, семантическим анализом, построением и оптимизацией плана выполнения запроса, выполняются на этапе компиляции. В момент исполнения этого оператора СУБД просто изымает хранимый план выполнения этого оператора и исполняет его.

В случае динамического SQL ситуация абсолютно иная. На момент компиляции мы не видим и не знаем текст оператора SQL и не можем выполнить ни одного из четырех обозначенных этапов. Все этапы СУБД должна будет выполнять без предварительной подготовки в момент исполнения программы.

Условные временные диаграммы выполнения SQL-операторов в статическом SQL и в динамическом SQL. Конечно, динамический SQL гораздо менее эффективен в смысле производительности, по сравнению со статическим SQL. Поэтому во всех случаях, когда это возможно, необходимо избегать динамического SQL. Но бывают случаи, когда отказ от динамического SQL серьезно усложняет приложение. Например, в случае с поиском по произвольному множеству параметров невозможно заранее предусмотреть все возможные комбинации запросов, даже если возможных параметров два десятка. А если их больше, то именно динамический SQL становится наиболее удобным методом решения необъятной проблемы.

Наиболее простой формой динамического SQL является оператор непосредственного выполнения EXECUTE IMMEDIATE. Этот оператор имеет следующий синтаксис:

EXECUTE IMMEDIATE <имя_базовой переменной>

Базовая переменная содержит текст SQL оператора.

Однако оператор непосредственного выполнения пригоден для выполнения операции, которые не возвращают результаты.
Так же как в статическом SQL, для работы с множеством записей вводится понятие курсора и добавляются операторы по работе с курсором, и в динамическом SQL должны быть определены подобные структуры.
Прежде всего было предложено разделить выполнение SQL-оператора в динамическом SQL на два отдельных этапа.
Первый этап называется подготовительным, он фактически включает 4 первых этапа выполнения SQL-операторов, рассмотренные нами ранее: синтаксический и семантический анализ, построение и оптимизация плана выполнения оператора.

Этот этап выполняется оператором PREPARE, синтаксис которого приведен ниже:

PREPARE <имя_оператора> FROM <имя_базовой переменной>

<имя_оператора> - это идентификатор базового языка.

На втором этапе этот определенный на первом этапе оператор может быть выполнен операцией EXECUTE, которая имеет синтаксис:
EXECUTE <имя__оператора> USING {<список базовых переменных> |
DESCRIPTOR <имя_дескриптора>}
Здесь DESCRIPTOR — это некоторая структура, которая описывается на клиенте, но создается и управляется сервером. Дескриптор представляет совокупность элементов данных, принадлежащих СУБД. Программное обеспечение СУБД должно содержать и поддерживать набор операций над дескрипторами. Эта структура была введена в стандарт SQL2 для типизации динамического SQL.
В стандарт SQL2 введены следующие операции над дескрипторами:
ALLOCATE DESCRIPTOR <имя_дескриптора> [WITH MAX <число_элементов>] — оператор связывает имя дескриптора с числом его базовых элементов и обеспечивает выделение памяти под данный дескриптор.

DEALLOCATE DESCRIPTOR <имя_дескриптора> — оператор освобождает разделяемую память СУБД, занятую хранением описания данного дескриптора. После выполнения данного оператора невозможно обратиться к дескриптору ни с одной операцией.
SET DESCRIPTOR {COUNT = <имя_базовой переменной> VALUE <номер элемента> {<имя_ элемента>= <имя_базовой переменной>[...]}} — оператор занесения в дескриптор описания передаваемых параметров. Описания перелаются СУБД, которая их обрабатывает, внося соответствующие изменения в область данных, отведенную под дескриптор.
GET DESCRIPTOR { <имя_базовой переменной> = COUNT | VALUE <номер элемента> {<имя_базовой переменной>=<имя_элемента>[...]} — оператор получения информации из дескрипторов после выполнения запроса.
DESCRIBE [INPUT | OUTPUT] <имя_оператора> USING SQL DESCRIPTOR <имя_дескриптора> — оператор, позволяющий получить описания таблиц результатов запросов (DESCRIBE OUTPUT) или входных параметров (DESCRIBE INPUT).

OPEN <имя_курсора> [USING <список базовых переменных> | USING SQL DESCRIPTOR <имя_дескриптора>] — динамический оператор открытия курсора.

FETCH <имя_курсора> [USING <список базовых переменных> | USING SQL DESCRIPTOR <имя_дескриптора>] — динамический оператор перемещения но курсору.

DEALLOCATE PREPARE <имя_оператора> — оператор уничтожает ранее подготовленный план выполнения оператора SQL и освобождает разделяемую память СУБД, связанную с храпением этого плана. Этот оператор имеет смысл применять, если не применять команду выполнения к подготовленному ранее оператору SQL.

Следует отметить, что в настоящий момент большинство СУБД реализуют динамический SQL несколько отличными от стандарта способами, однако в ближайшем будущем все поставщики вынуждены будут перейти к стандарту, так как именно это привлекает пользователей и делает переносимым разрабатываемое прикладное программное обеспечение.