Database Primer

Entity Relationships

One-to-many relationship

We have a company which manages multiple software projects. We hire people and assign them to projects. Consequently, each project will have several people working on it. From this statement, it is easy to observe that project-to-workers constitute a one-to-many relationship, i.e. multiple people work on one project.

Note that in this relationship, we are assuming that one person may not work on more than one project.

If we had to represent a one-to-many relationship in the relational database, it would do it like this:

Table: EMPLOYEE

ID	First	Last	SSN	PROJECT_ID	...
1	One	First	111-11-1111	1	
2	Two	Second	222-22-2222	1	
3	Three	Third	333-33-3333	2	
4	Four	Forth	444-44-4444	2	
5	Five	Fifth	555-55-5555	2	
6	Six	Sixth	666-66-6666	3	
7	Seven	Seventh	777-77-7777	3	
8	Eight	Eighth	888-88-8888	4	
9	Nine	Ninth	999-99-9999	4	

Table: PROJECTS

ID	Name	...
1	P-1	
2	P-2	
3	P-3	
4	P-4	
...	...	...

Notes:

1. We only included a minimum set of fields to demonstrate the point of the current topic.

Let us analyze this design.

Each table **must** have a unique primary key. In both EMPLOYEE table and PROJECT tables, this field is named "ID". There are times when the primary key is made up of multiple keys, referred to as "composite key", however, we will not consider this case here.

For the EMPLOYEE table, we could have designated the SSN as the primary key since we know that the Social Security number would be unique. This would not be a great idea for at least two reasons:

1. We may hire some foreign workers who do not have a Social Security number and whatever unique ID they may have, will likely not conform to the SSN format - "xxx-xx-xxxx".
2. If we chose SSN to be the primary key, its datatype would be STRING. Primary keys are used for joining data from multiple tables. Joining on a field whose type is INTEGER is a lot faster than if its type were STRING.

All databases have a SEQUENCE GENERATOR. We can designate the primary key as a SEQUENCE GENERATED key and the database will created it for us. Thus, when we try to insert our first employee into the PEOPLE table, it's ID will be 1.

From EMPLOYEE and PROJECT tables above we can see that both have a primary key named "ID".

To express the one-to-many relationship, the EMPLOYEE table also has a key named "PROJECT_ID". The PROJECT_ID is a foreign key. A foreign key is the primary key of another table, in this case table PROJECT.

From the demonstration above, we can see that people whose IDs are 1 & 2 work for project ID=1, people with IDs 3, 4, and 5 work for project ID=2, etc.

The one-to-many (or many-to-one) relationships are incredibly common.

Consider another situation where we want to store information about each employee. In addition to the obvious fields such as "First Name", "Last Name", "SSN", each employee also has a phone number, email address, and residence address. However, many employees will have more than one phone number, email address, and may have multiple residences. The only logical way to represent this would be with a one-to-many relationship:

EMPLOYEE

ID	...
1	..
2	..
3	..
..	..

PHONES

ID	PHONE	EMPLOYEE_ID
1	555-1212	1
2	123-1234	1
3	234-9876	2
4	767-9878	2
..	..	..

EMAILS

ID	EMAIL	EMPLOYEE_ID
1	..	1
2	..	3
..	..	..

ADDRESSES

ID	ADDRESS	EMPLOYEE_ID
1	..	2
2	..	2
..	..	..

One-to-one relationship

One-to-one relationships are less common than many-to-one but they are still used very extensively. Lets go back to the task of storing data for each employee. In addition to storing employee's name, phone-numbers, etc., we also want to store employee's salary, start and end date (end data would be set to null for current employees), and other sensitive information. SSN field may also be considered sensitive data. While many applications will need access to employee's general information, such as name, department number, title, etc., only HR applications that are entrusted to few employees should have access to sensitive data, such as employee's salary.

Each database table can have associated privileges indicating who can see and modify data. Thus, we can create other tables for storing employee's sensitive data and assign privileges to this table to select few.

The one-to-one relationships are represented the same as the many-to-many relationships, that is with a foreign key.

Table: EMPLOYEE

ID	First	Last	...
1	One	First	..

Table: EMPLOYEE_HR

ID	EMPLOYEE_ID	SSN	SALARY	...
1	1	111-11-1111	120000	..

2	Two	Second	...	2	2	222-22-2222	150000	...
3	Three	Third	...	3	3	333-33-3333	240000	...
...								

Note:

1. EMPLOYEE_ID in EMPLOYEE_HR table is the foreign key. It contains values from EMPLOYEE.ID field.
2. Unlike the primary key, a foreign need need not (and usually will not) be unique and it may be null.
3. In the case of a one-to-one relationship, we could have placed the "foreign" key in either table.

For example, our design could look like this, where we place the foreign key - EMPLOYEE_HR_ID in the EMPLOYEE table.

Table: EMPLOYEE

ID	First	Last	EMPLOYEE_HR_ID
1	One	First	1
2	Two	Second	2
3	Three	Third	3
...			

Table: EMPLOYEE_HR

ID	SSN	SALARY	...
1	111-11-1111	120000	...
2	222-22-2222	150000	...
3	333-33-3333	240000	...

However, since most people in the organization will be privy to data in the EMPLOYEE table, and only people who work in HR will be able to see the EMPLOYEE_HR table, it is better to place the foreign key into the EMPLOYEE_HR table so that those who examine contents of the EMPLOYEE table would not even know that EMPLOYEE_HR table exists.

There are other reasons to create one-to-one relationships.

Let's say we also want to store a single hobby for our employees. The description of a hobby may contain many fields. Most employees are very boring and do not have any hobbies. If we added Hobby fields to the EMPLOYEE table, most records would contain "holes", i.e. empty values. Thus, we can make a decision to define a HOBBY table and link it to EMPLOYEE with a one-to-one relationship.

Many-to-many relationship

Next, we expand our original definition by allowing one person to work on many projects. For example, some person may dedicate 60% of his time working on project #1 and 40% working on project #2. Any one person may work on more than two projects. Following this rule, we progress from a one-to-many relationship to a many-to-many relationship. In this many-to-many relationship, we have many people working on one project and each person may work on many projects. To support the many-to-many relationship, we need to introduce a new "MAPPING" table.

Note that table PROJECT_WORKER_MAP, maps Projects to Employees (Workers). For example,

Table: PROJECT

ID	Name	...
1	P-1	
2	P-2	
3	P-3	
4	P-4	
5	P-5	

TABLE: PROJECT_EMPLOYEE_MAP

ID	EMPLOYEE_ID	PROJECT_ID
1	1	1
2	2	1
3	2	2
4	4	2
5	5	2
6	1	3
7	7	3
8	8	3
9	8	4

TABLE: EMPLOYEE

ID	First	Last	...
1	One	First	
2	Two	Second	
3	Three	Third	
4	Four	Forth	
5	Five	Fifth	
6	Six	Sixth	
7	Seven	Seventh	
8	Eight	Eighth	
9	Nine	Ninth	

From the PROJECT_EMPLOYEE_MAP table we can see that employees with IDs 1 and 2 work on project 1, employee with IDs 2, 4, and 5 work on project 2, employees with IDs 1, 7, and 8 work on project 3, etc. We can also observe that employee with ID 2 works on projects 1 and 2, employee with ID 1 works on projects 1 and 3.

With many-to-many relationships, it is quite possible to have some employees who do not work on any projects, let's say that an active employee goes on a sabbatical and will not be involved on any project. It is also possible to have a project that does not have any employees, for example, an upstart project that is still in the research phase. In cases where employees are not assigned to any projects or project do not have any employees, the PROJECT_EMPLOYEE_MAP table will not have a row for associating employee to a project.

Note that in the example above, no employee is assigned to project ID=5 and Employee ID=9 does not work on any projects.

Database Commands - DDL and DML

All databases support two types of commands: DDL that stands for Data Definition Language and DML that stands for Data Manipulation Language.

DDL Commands

The list of DDL commands is exhaustive. You use DDL commands to created and alter tables and indexes, drop (i.e. delete) tables, manage privileges, and much-much more.

Create Table

Before we can execute any database commands we need to create required tables and populate them with data. Let us examine the CREATE TABLE command:

```
create table worker (
    id integer not null,
```

```
    created_by varchar(255),
    in_utc timestamp not null,
    last_modified_by varchar(255),
    out_utc timestamp,
    audit_id varchar(255),
    compensation double not null,
    worker_code varchar(255),
    worker_name varchar(255),
    primary key (id)
);
```

The WORKER table in our real project has following fields:

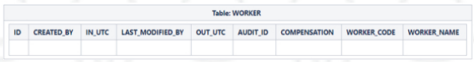


Table: WORKER

ID	CREATED_BY	IN_UTC	LAST_MODIFIED_BY	OUT_UTC	AUDIT_ID	COMPENSATION	WORKER_CODE	WORKER_NAME

Each field is listed along with its datatype. For STRING fields (i.e. varchar) we also need to specify maximum length.

Some fields may require a value, i.e. may not contain nulls. For example, "compensation double not null"

One or more fields must be designated as primary. (primary key (id)). A Primary key in unique and non-nullable by definition.

Indices and Unique Indices

Databases support many types of indices and we can't cover this topic exhaustively. Let's first discuss an index in a general sense.

Suppose we have an ORDERS table with 1,000,000 rows. We may want to perform certain types of operations on the ORDERS table very frequently. For example, find order where SELLER_NAME is "One First". With 1M rows, we would have to check 1M records in the worst case (order O(n)).

```
SELECT *
FROM ORDERS
WHERE SELLER_NAME = 'One First'
```

However, if we create an index for field SELLER_NAME, we would be able to locate this record in almost constant time (O(1)).

The DDL command to create an index may look like this:

```
CREATE INDEX INDEX_SELLER_NAME
ON ORDERS(SELLER_NAME)
```

One of the common interview questions goes something like this:

"You work on your application and test it continuously in your dev (development) environment. Everything works great so you decide to deploy it to the production environment. However, in production, your application's performance becomes very sluggish. Why is that?"

Your answer may be as follows:

"It is quite possible that the dev database has been property tuned with required indices and it is therefore performant. However, production environments uses a different database and it is possible that it does not have all the required indices".

In addition to a simple index, databases support a notion of a unique index.

Let's say that we want our database to enforce business rules and one of our business rules is that the combination of values in audit_id and out_utc fields must be unique. Thus, it would be an error if two records had same values in both audit_id and out_utc fields. The DDL command for a unique index would look like this:

```
create unique index UKbe6uin1lttt8dh3oddmrb6g6d on worker (audit_id, out_utc);
```

The name of the index ("UKbe6uin1lttt8dh3oddmrb6g6d" in the example above) was randomly generated and the name itself is not important. The only thing that is important is to identify a set of fields that must be unique.

Constraints

Let's go back to our many-to-many illustration and consider what would happen if someone decided to delete an employee with ID=1

```
delete from employee where id = 1
```

We would be left with records in table PROJECT_EMPLOYEE_MAP that would not make sense. (See records with ID 1 and 6 where the EMPLOYEE_ID is 1 which was deleted)

Table: PROJECT

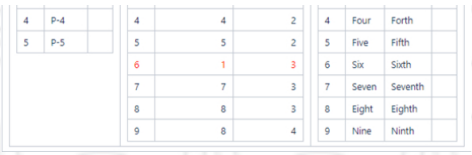
ID	Name	...
1	P-1	
2	P-2	
3	P-3	

TABLE: PROJECT_EMPLOYEE_MAP

ID	EMPLOYEE_ID	PROJECT_ID
1	1	1
2	2	1
3	2	2

TABLE: EMPLOYEE

ID	First	Last	...
1	One	First	
2	Two	Second	
3	Three	Third	



4	P-4	
5	P-5	

4	4	2
5	5	2
6	1	3
7	7	3
8	8	3
9	8	4

4	Four	Forth
5	Five	Fifth
6	Six	Sixth
7	Seven	Seventh
8	Eight	Eighth
9	Nine	Ninth

We can still delete employee with ID=1, however, we first need to delete records from the PROJECT_EMPLOYEE_MAP table that associate employee ID=1 with projects. This constraint can be enforced with command:

```
alter table project_worker_map add constraint FK6baomleyjvq2jkfy8n5tscn0l foreign key (worker_id) references worker;
```

There are too many DDL commands and it is not important to know them all, only to know that they exist and can be easily discovered. Most of the common DDL commands affect tables and indices, such as:

```
create table ...

alter table ...

drop table ...

create index ...

create unique index ...

drop index ...
```

Privileges

We had mentioned earlier the need for a one-to-one relationship. In that example, we decided to store sensitive employee information inside EMPLOYEE_HR table, accessible to HR staff only. For people working in HR who do need this access, we can assign them to a user (or group) with a special name, i.e. USER_HR. Then, we can execute the following command:

```
GRANT ALL ON TABLE EMPLOYEE_HR TO USER USER_HR
```

One may think that knowing the syntax of this command might allow you to peek at other employee's salary. However, only database admins, or other users with special privileges, would be able to execute the GRANT command.

DML Commands

The good news about Data manipulation commands (DML) is that they comprise only four commands: INSERT, UPDATE, DELETE, and SELECT. In the industry, these are often referred to as CRUD commands for Create, Read, Update, Delete.

The bad news is that some of these commands, particularly SELECT could get very complex. In this document we will only cover the basics.

INSERT Command

Most of the INSERT commands are very simple.

For example:

```
INSERT INTO WORKER(ID, AUDIT_ID, IN_UTC, OUT_UTC, WORKER_CODE, WORKER_NAME, COMPENSATION, CREATED_BY, LAST_MODIFIED_BY)

VALUES(1, '00000000-0000-0000-0000-000000000001', CURRENT_TIMESTAMP - CURRENT_TIMEZONE, '9999-12-31', 'CODE-1', 'One First', 120000.00, 'unknown', 'unknown');
```

This example could also be written like this:

```
INSERT INTO WORKER VALUES(1, '00000000-0000-0000-0000-000000000001', CURRENT_TIMESTAMP - CURRENT_TIMEZONE, '9999-12-31', 'CODE-1', 'One First', 120000.00, 'unknown', 'unknown');
```

The reason why the above syntax would work is because we are populating every single field in the WORKER table. If we only wanted to populate some of the fields, we would need to list them, as in:

```
INSERT INTO WORKER(ID, AUDIT_ID, WORKER_CODE, WORKER_NAME, COMPENSATION)

VALUES(1, '00000000-0000-0000-0000-000000000001', 'CODE-1', 'One First', 120000.00);
```

Note that if you don't specify a field inside the INSERT statement, that field must be nullable.

UPDATE Command

To change values in an existing record we can do this:

```
update worker

set compensation = 200000,

worker_code = 'CODE-X'

where id = 3
```

Naturally, this syntax could get a lot more complex and we will return to this command later.

DELETE Command

The simplest delete command looks like this:

```
delete from worker
```

This command will delete all records from table WORKER.

One of the common interview questions it to ask about the difference of this `delete` command and the `drop table` command:

DELETE	DROP
delete from worker	drop table worker

When you use the `delete` command, the table still exists but contains no records. When you `drop` the table, the table is destroyed and you would need to create it again if you decide to add records to it.

More often than not, we will add a condition to the delete command, for example:

Command	Description
delete from worker where compensation < 100000	Delete workers who make less than 100,000
delete from worker where worker_code in ('CODE-1', 'CODE-2')	Delete workers with WORKER_CODE CODE-1 and CODE-2

SELECT Command

This is where all the fun is. Many books and PhD decertations are dedicated to this topic. Let us start simple:

Note:

1. To select all fields use "`select * ...`"
2. We use "`where`" to filter records to be displayed. Multiple expressions may be strung together with `and, or, in,` and other operators.
3. Note the usage of `like` operator
4. Aliases are often used with SELECT commands.

GROUP BY and HAVING Clauses

Another powerful feature of SELECT is to aggregate records via GROUP BY. Let's start with example. We would like to find out how many people work on each project. We use the PROJECT_WORKER_MAP table to map workers to projects:

Query	Response
select project_id, count(worker_id) as WorkerCount from project_worker_map group by project_id	PROJECT_ID WORKERCOUNT ---------- 1 2 2 2 3 4

Suppose we are interested in seeing only those projects where the worker count is greater than 2

Query	Response
select project_id, count(worker_id) as Worker_Count from project_worker_map group by project_id having count(worker_id) > 2	PROJECT_ID WORKER_COUNT ---------- 3 4

Note:

having to group by is what where is to select

Above command is incredibly useful and you should store it somewhere in your memory bank. Often you run a process which will populate rows in some table. It may be an error to have records with a duplicate value in some field. Thus, you would execute the following query to check:

Query
select some_field from some_table group by some_field having count(some_field) > 1

Joining Tables

Let us take a leap into something more interesting. We would like to determine the total compensations for each project. The compensation is specified in the WORKER table, so we need to join WORKER and PROJECT_WORKER_MAP tables.

Query	Result
select p.project_id, sum(w.compensation) as TotalComp from worker w, project_worker_map p where = p.worker_id group by project_id	PROJECT_ID TOTALCOMP ---------- 1 310000.0 2 425000.0 3 735000.0

Let us observe several things from above command:

In group by selects, fields that are listed on the select clause must match fields that appear in the group by clause. However, the select clause may also specify aggregate operations such as SUM, AVG, and others.

As a slight variation to above, let's say we only want to see projects whose total compensation is greater than 500,000.

Query	Result
select p.project_id, sum(w.compensation) as TotalComp from worker w, project_worker_map p where = p.worker_id group by project_id having sum(w.compensation) > 500000	PROJECT_ID TOTALCOMP ---------- 3 735000.0

Notes:

In the previous query, I used alias w for worker and p for project_worker_map. If I had not used aliases, I would need to type the full name of the table, as in:

```
select project_worker_map.project_id, sum(worker.compensation)
from worker, project_worker_map
where worker = project_worker_map.worker_id
group by project_id
having sum(worker.compensation) > 500000
```

Min and Max

These examples will be provided without explanations:

Query	Response
select max(compensation) as Max from worker	MAX -------- 240000.0
select min(compensation) as Min from worker	MIN ------- 40000.0

Subqueries

Suppose we only have information on WORKER_CODE, i.e. 'CODE-1' and 'CODE-2' and we would like to find out what projects they are working on. To join table WORKER to table WORKER_PROJECT_MAP we need to have WORKER.ID and WORKER_PROJECT_MAP.WORKER_ID. However, we do not know values for WORKER.ID, we only know WORKER_CODE.

Before we accomplish our final task, let's perform some intermediate steps, like joining three tables: PROJECT, WORKER, and PROJECT_WORKER_MAP.

Query	Response
select p.project_name, w.worker_name from project p, worker w, project_worker_map m where m.project_id = p.id and m.worker_id = w.id	PROJECT_NAME WORKER_NAME -------------------- P-1 One First P-1 Three Third P-2 Two Second P-2 Four Forth P-3 One First P-3 Four Forth P-3 Three Third P-3 Two Second
select p.project_name, w.worker_name from project p, worker w, project_worker_map m where m.project_id = p.id and m.worker_id = w.id order by worker_name	PROJECT_NAME WORKER_NAME -------------------- P-2 Four Forth P-3 Four Forth P-1 One First P-3 One First P-1 Three Third P-3 Three Third P-2 Two Second P-3 Two Second

Note:

Second command uses `order by` clause to sort results by `worker_name` field.

However, for our current task, we don't have IDs of workers, we have `worker_code`. Thus, we need to use the worker_code to get to the WORKER.ID.

Query	Result
select p.project_name, w.worker_name from project p, worker w, project_worker_map m where m.project_id = p.id and m.worker_id = w.id and m.worker_id in (select id from worker where worker_code in('CODE-1', 'CODE-2'))	PROJECT_NAME WORKER_NAME -------------------- P-1 One First P-2 Two Second P-3 One First P-3 Two Second

Note:

We need to join three tables and define the filtering criteria

We can combine this result with GROUP BY clause to determine the total compensation for these projects:

Query	Result
select p.project_name, sum(w.compensation) as TotalComp from project p, worker w, project_worker_map m where m.project_id = p.id and m.worker_id = w.id and m.worker_id in (select id from worker where worker_code in('CODE-1', 'CODE-2')) group by p.project_name	PROJECT_NAME TOTALCOMP ---------- P-1 120000.0 P-2 240000.0 P-3 360000.0

Next Steps

The information presented so far does not even scratch the surface. There are things that you need to be aware of even though they are not demonstrated in this document.

SQL Standards

Like everything else, SQL went through a lot of iterations over many years. The SQL presented here follows the original SQL-86 format. I believe the latest is called SQL:2003. The syntax for SELECT commands has changed, however, the original syntax is still supported. Some operations differ by database providers. For example, to perform a left outer join (discussed shortly), you will need to adhere to the format of the database provider such as Oracle, Sybase, Postgres, DB2, etc.

The main point is to understand what capabilities are possible. It is easy to find the syntax for a specific command on the internet.

LEFT and RIGHT OUTER JOINS

So far, the examples provided here used a standard, or "INNER" join. For example, we can join multiple tables using INNER join to see all projects and people who are assigned to them. Some projects may not have any people assigned. Suppose we want to see all projects. If people are assigned to them, we will see person name in the WORKER column. If a project has no workers assigned, that project will not appear in the response. Often, we want to see projects and workers assigned to them and also projects that have no workers. This is the job for a LEFT OUTER join. If no one is assigned to some project, the project will still appear but the value in the WORKER column will be null. RIGHT OUTER join is similar to LEFT OUTER join but works in the right-to-left order.

SELF JOIN

Here is one of the most common problems for a SELF JOIN. Let's say we have a table of employees, along with their salaries and ID of the manager. Let's say our fictitious table looks like this:

Table: EMPLOYEE

ID	Name	Salary	MANAGER_ID	...
1	One First	50000	5	
2	Two Second	70000	5	
3	Three Third	90000	5	
4	Four Fourth	180000	5	
5	Five Fifth	160000	9	
...	....	...	...	..

We need to answer following question: "Provide a list on employees who make more money than their manager." From table's data we can observe that Employee IDs 1, 2 3, 4 all report to employee whose ID is 5. Only employee ID=4 makes more money than his manager.

This question comes up so often at interviews that I owe you a demonstration.

Let's create a new table with data as shown above:

Create Table	create table workerX (id integer not null, name varchar(255), compensation double not null, manager_id integer not null)
Populate	insert into workerX (id, name, compensation, manager_id) values (1, 'One First', 50000, 5) insert into workerX (id, name, compensation, manager_id) values (2, 'Two Second', 70000, 5) insert into workerX (id, name, compensation, manager_id) values (3, 'Three Third', 90000, 5) insert into workerX (id, name, compensation, manager_id) values (4, 'Four Fourth', 180000, 5) insert into workerX (id, name, compensation, manager_id) values (5, 'Five Fifth', 160000, 9)

Now we can execute a query to find all employees who make more money than their managers:

Query	Result
select employee.* from workerX employee, workerX manager where employee.manager_id = and employee.compensation > manager.compensation	ID NAME COMPENSATION MANAGER_ID ---------- 4 Four Fourth 180000.0 5

Note how use of aliases, employee and manager, were absolutely essential for a SELF JOIN query.

UNIONS

There may be times when we would like to combine results from multiple tables. Our SELECT clause identifies fields and they datatype. As long as the fields in the SELECT clause match, data from multiple tables may be combined with a UNION command.

Stored Procedures, Triggers, and Temporary Tables

SELECT command is extremely powerful and provides many clauses to wield incredible power. In addition to standard features of the SQL, each database provide furnishes a library of functions to works with strings, dates, and just about anything else. However, sometimes all this power is still not enough. There may be times when it would be useful to create intermediate results and stored them in the temp table. Next, subsequent commands can me used in combination with data in the temp table to produce more elaborate answers.

For anything that can't be done with SQL, there are stored procedures. Each database provider has its own programming language to interact with databases. A program written in this language is called the stored procedure. Some companies build their entire applications with stored procedures. In my view, this is a terrible idea but it may be justified for certain types of projects. One of the problems if stored procedures is that they can't be debugged. You may need tp place results of some intermediate computation into a table for debugging purposes.

It is also important to realize that a stored procedure is often composed to several SQL statements without resorting to any other programming aspects. To support transactions, you will need to place your SQL commands inside the stored procedure.

A Trigger is a stored procedure. It is triggered when a certain action occurs, such as inserting a new row or updating it. Some time ago, triggers where relied on very heavily, however, over the years they usefulness has diminished.

Transactions and RollBacks

Let's say we have a site for taking customer orders. When the order is placed we need to update ORDERS table, INVENTORY table, and possible CUSTOMERS table to keep track of all the money they had spent on our crap.

Say we update the ORDERS table first and then try to update the INVERTORY table, but that update fails. Now we have a system in an inconsistent state. To prevent this inconsistent state we need to ROLL-BACK our transaction. Thus, either all commands complete successfully or none at all.

Failure to update the INVERNTORY table may happen for many diffident reasons, for example, the table may run out of storage space that was allocated for it.