Database Design for Black Country Bespoke Furniture Ltd

# Introduction

This document has been prepared for the client, Black Country Bespoke Furniture (BCBF). It contains the design and implementation script for the database of a new customer order management system.

# Organisation of this document

The following sections which have been put together in accordance with industry standard practices:

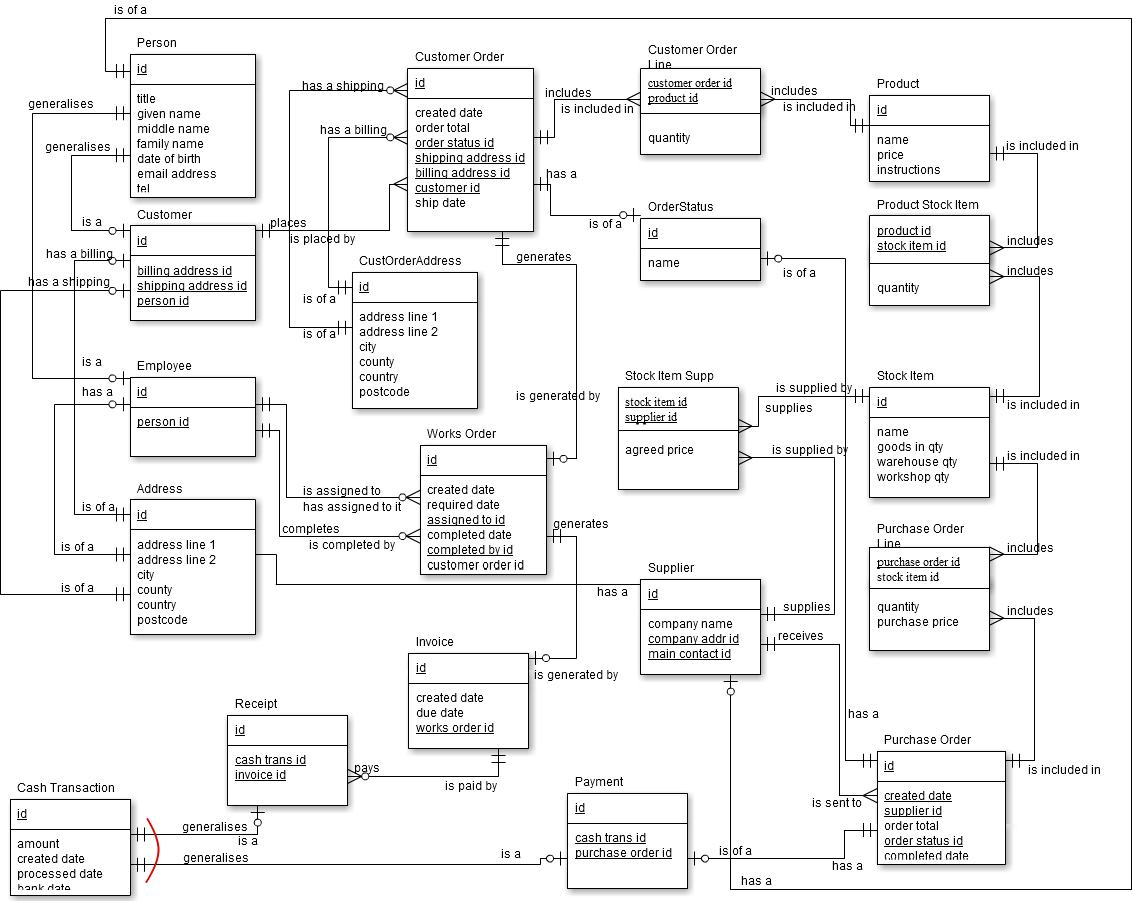
1. Entity Relationship Model



* 1. Entity Relationship Diagram
  2. Set of Relations in Boyce-Codd Normal Form annotated with design choices
  3. Design choices for database optimisations such as clusters and partitions

1. Roles and privileges
   1. Design choices with respect to Role Based Security
   2. Listing of security roles
   3. Listing of logical views and their design choices
2. Appendices
   1. Screenshots of various query examples
   2. SQL scripts for creating and populating the database

# Entity Relationship Diagram



Key:

|  |  |
| --- | --- |
|  | One and only one |
|  | Zero or one |
|  | One or more |
|  | Zero or more |
|  | XOR (exclusive or) |

# Relations, Assumptions and Design Choices

For clarity, this section contains a standard section for each Entity in the above diagram. Each section contains:

* A BCNF conformed relation which shows:
  + The name of the Entity
  + All the necessary attributes
  + The attributes’ types
  + The attributes’ domains
  + The attributes’ constraints
* Excerpt from the customer brief which justifies the reason to include the object
* A short paragraph explaining any assumptions and design choices

|  |  |  |  |
| --- | --- | --- | --- |
| **Person** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Title | Character | Max Length 10, NULL | { Mr, Mrs, Ms, Miss, Dr, Lord, Rt. Hon, Sir, Gen., HRH, … } |
| Given Name | Character | Max length 20, NOT NULL, Only alpha first position, then alpha and limited special chars. No numbers. | Regex representation:  ^[a-zA-Z][a-zA-Z -'.]{0,19}  *Note: single letter allowed* |
| Middle Name | Character | Max length 20, NULL, Only alpha first position, then alpha and limited special chars. No numbers. | Regex representation:  ^[a-zA-Z][a-zA-Z -'.]{0,19}  *Note: single letter allowed* |
| Family Name | Character | Max length 20, NOT NULL, Only alpha first position, then alpha and limited special chars. No numbers. | Regex representation:  ^[a-zA-Z][a-zA-Z -'.]{1,19} |
| Date of Birth | Date | INDEX |  |
| Email Address | Character | UNIQUE, Max Length 50, Regular Expression | Regex representation:  [a-zA-Z0-9.\_%-]+@[a-zA-Z0-9.\_%-]+\.[a-zA-Z]{2,4} |
| Tel | Character | Max Length 20, NULL, Only numbers and +, -, () and space |  |

Justification from brief: NA – this is an inferred super-type of other objects.

Person is a super-type of Customer, Employee and Supplier Main Contact. It holds common personal information such as name, date of birth and email address. Each of these requires important constraints to avoid garbage entries. This inheritance model allows the constraints to be specified in one place and re-used, while also allowing an Employee or a Supplier Main Contact to also be a Customer without entering their details multiple times.

In case previously unforeseen Person details are required, these constraints can be easily modified by issuing an ALTER TABLE statement. Only the Sales Administrator role will be given permission to do this and the role only issued to a trusted employee.

Ideally, date of birth should be constrained to disallow unrealistic dates. However, as this is a moving target it is expected this will be validated in a user interface/front-end. Furthermore, date of birth will be indexed to improve the efficiency of a specific query where they marketing department want to create runs of birthday cards for customers each week.

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Address Line 1 | Character | Max Length 50, NOT NULL |  |
| Address Line 2 | Character | Max Length 50, NULL |  |
| townOrCity | Character | Max length 20, NOT NULL |  |
| County | Character | Max length 20, NULL |  |
| Country | Character | Max length 20, NOT NULL, Default ‘United Kingdom’ | Should be a constrained list of countries / territories, perhaps as a look up table with FK relationship (not implemented) |
| PostCode | Character | Max length 10, NULL | NULLs allowed, e.g. no postcodes in Republic of Ireland. |

Justification from brief:

“*The Customer Order consists of the customer’s name, the delivery address, the billing address…*”

Address set up as its own entity to remove repeating groups from Customer entity (billing and shipping addresses). It holds common address information such as Address Line 1, City, Post Code. Some of these require NOT NULL constraints to avoid garbage entries. This model allows the constraints to be specified in one place and re-used in multiple relationships such as Customer Order Shipping Address and Supplier Company Address.

A nested table was also considered as a useful way to model addresses. The benefit of a nested table is that the instance used in each location is completely separate by design. Unfortunately, it is complicated to add constraints to this type of object so this idea was abandoned.

As billing and shipping address is also used in CustomerOrder, it is highly desirable to use a single entity to model all the address with foreign key relationships. However, it is necessary to ensure that addresses used in Customer Orders are separate from the originating customer and are never updated by a Customer change of address which would cause inconsistent historical data. Therefore, an Address table and a separate CustomerOrderAddress table are created.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| billingAddressId | Integer | Foreign Key, NOT NULL | Only existing Address Primary Key Values |
| ShippingAddressId | Integer | Foreign Key, NOT NULL | Only existing Address Primary Key Values |
| personId | Integer | UNIQUE, Foreign Key, NOT NULL | Only existing Person Primary Key Values |

Justification from brief:

“*A Customer places an order…*”

Customer is a sub-type of Person so a Customer instance will always be of one and only one Person. A Customer is only added to the system when the first order is placed so each customer will always have at least one Customer Order. It is assumed a Customer may have either one or two addresses. There will always be a billing address, and there may be a separate shipping address. If there is no separate shipping address, the billing address will be used for shipping. When a Customer Order is placed, it is also possible to change the shipping address per Customer Order, to allow the finished item to be sent to a different location (e.g. an office, friend’s or relative’s house). See also Customer Order and CustOrderAddress relations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Employee** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| personId | Integer | UNIQUE, Foreign Key, NOT NULL | Only existing Person Primary Key Values |

Justification from brief:

“*A Works Order consists of […] and who the job was assigned to.*”

Employee is a sub-type of Person so an Employee instance will always be of one and only one Person. Only basic details are stored in this database as it is not intended to replace the HR system. Employee is used to track which member of the manufacturing department completes a Works Order. An Employee (of the manufacturing dept) completes Work Orders, but a new Employee may not have completed any Work Order until they have received adequate training/supervision. An Employee may have zero (when they are new) or more Works Orders that they’ve been assigned and zero or more that they have completed.

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderStatus** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| StatusName | Char | UNIQUE, Length 10, NOT NULL |  |

Justification from brief:

“*The Customer Order consists of […] the order status (e.g. pending, confirmed, dispatched, paid)…*”

OrderStatus This is a look up table of status names to match the basic business process such as “Pending,” “Confirmed,” “Despatched” and “Paid.” The names are constrained to be unique to avoid data integrity issues.

|  |  |  |  |
| --- | --- | --- | --- |
| **CustOrderAddress** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Address Line 1 | Character | Max Length 50, NOT NULL |  |
| Address Line 2 | Character | Max Length 50, NULL |  |
| townOrCity | Character | Max length 20, NOT NULL |  |
| County | Character | Max length 20, NULL |  |
| Country | Character | Max length 20, NOT NULL, Default ‘United Kingdom’ | Should be a constrained list of countries / territories, perhaps as a look up table with FK relationship (not implemented) |
| PostCode | Character | Max length 10, NULL | NULLs allowed, e.g. no postcodes in Republic of Ireland. |

Justification from brief:

*(Inferred during the implementation process. Customers must be able to choose other addresses than their own for billing and shipping and these must be independent from the customer details which may change from time to time).*

“*The Customer Order consists of the customer’s name, the delivery address, the billing address…*”

CustOrderAddress stores billing and shipping addresses from Customer Orders. These will default to the customer specific details in any front end application but it must be possible to change them so that items can be purchased on behalf of other people and sent to alternative addresses. Various options have been considered to maintain the best referential integrity including Nested Tables and different kinds of triggers. However, the selected implementation of two separate tables provides the most intuitive and easy to maintain option.

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerOrder** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| createdDate | Date | NOT NULL, Default SYSDATE |  |
| shipDate | Date | Cannot be earlier than createdDate |  |
| orderTotal | Decimal(8,2) | NOT NULL, cannot be less than zero | Initially calculated from sum of order line totals. |
| orderStatusId | Integer | Foreign Key, NOT NULL | Only existing OrderStatus Primary Key Values |
| billingAddressId | Integer | Foreign Key, NOT NULL | Only existing CustOrderAddress Primary Key Values |
| shippingAddressId | Integer | Foreign Key, NOT NULL | Only existing CustOrderAddress Primary Key Values |
| customerId | Integer | Foreign Key, NOT NULL | Only existing Customer Primary Key Values |

Justification from brief:

“*A Customer places an order…*”

Customer Order holds header detail about the Customer Order. This includes the date created, the date shipped (which is only known after the whole order is complete). Also included is which customer placed the Order along with the Order Total. The order total is calculated when the order is first created and is stored redundantly in order to act as a check. Any front-end application can re-calculate the order line totals and compare it to this number to ensure data integrity. Total must be zero or more (zero allowed for promotions, discounts, gratuities).

The order status is also stored via the FK to the OrderStatus table. Other information includes order specific Billing and Shipping Address, which defaults to the Customer settings but can be changed per Customer Order. These address details are stored in their own table so that any changes to customer address don’t affect historical order data.

A CustomerOrder has zero (when new) or one WorksOrder associated to it.

|  |  |  |  |
| --- | --- | --- | --- |
| **WorksOrder** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| createdDate | Date | NOT NULL, Default SYSDATE |  |
| requiredDate | Date | NOT NULL, cannot be earlier than createdDate |  |
| assignedToId | Integer | Foreign Key, NOT NULL | Only existing Employee Primary Key Values |
| completedDate | Date | NULL, cannot be earlier than createdDate |  |
| completedById | Integer | Foreign Key, NULL | Only existing Employee Primary Key Values |
| customerOrderId | Integer | UNIQUE, Foreign Key, NOT NULL | Only existing CustomerOrder Primary Key Values |

Justification from brief:

“*…sent to the manufacturing department which then makes the product from a Works Order…*”

Works Order is generated after a Customer Order has been received and the manufacturing department assigns an employee to it. A Works Order cannot exist without an originating Customer Order. There is always one and only one Customer Order linked to a Works Order.

Accepting the Customer Order and selecting the Employee to complete it is a management task and is not modelled here.

Works Order has information about which Employee is assigned to it and other information such as received date, completed date and so on. There is a separate completedById field to track cases where the Works Order is completed by a different employee than the one assigned. This is a NULLable foreign key as completedById is not known on creation of the Works Order.

Information stored on the Customer Order is not stored redundantly on the Works Order. When the products and stock items need to be looked up, a suitable join query is used. A Works Order will have zero (before completion) or one (after completion) Invoices linked to it.

|  |  |  |  |
| --- | --- | --- | --- |
| **Invoice** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| createdDate | Date | NOT NULL, Default SYSDATE |  |
| dueDate | Date | NOT NULL, Default SYSDATE + 30 days |  |
| worksOrderId | Integer | UNIQUE, Foreign Key, NOT NULL | Only existing CustomerOrder Primary Key Values |

Justification from brief:

“*Customer Order consists of […] date the invoice was sent to the customer…*”

Invoice is generated once the Works Order is completed. An Invoice cannot exist without an originating Works Order. An Invoice has one and only one Works Order linked to it.

Information stored on the Customer Order and Works order is not stored redundantly on the Invoice. When the order totals and order lines need to be looked up, a suitable join query is used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Product** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Name | Character | UNIQUE, Max Length 100, NOT NULL |  |
| Price | Decimal(8,2) | NOT NULL, cannot be less than zero |  |
| Instructions | Character | Max Length 250, NULL |  |

Justification from brief:

“*A Works Order consists of what Product is required […] the instructions for making the Product…*”

Product describes the standard product range available for manufacture and purchase. For the purposes of this project, the word Bespoke in the company name is taken to mean that standard items are “made to order” rather than unique items designed and built for each customer. A company working to this model would avoid bulky storage and inventory. A Product has a Name which identifies it uniquely and completely (e.g. Oxford Armchair - Green Velvet, Leather Footrest – Natural Cow Hide). There are no sub-types, categories or materials as it’s all captured in the name. Products are made up of various Stock Items and also come with instructions on how to assemble them from base Stock Items. Prices must not be less than zero but zero is allowed for items that are “included” with other items.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer Order Line** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| CustomerOrderId | Integer | Part of composite Primary Key with ProductId, Foreign Key, NOT NULL | Only existing CustomerOrder Primary Key Values |
| ProductId | Integer | Part of composite Primary Key with CustomerOrderId, Foreign Key, NOT NULL | Only existing Product Primary Key Values |
| Quantity | Integer | NOT NULL, cannot be less than one | { 1, 2, …, n } |

Justification from brief:

*(Inferred relation to represent the many to many join between Customer Order and Product)*

“*The Customer Order consists of […] the product(s) ordered, the quantity of each product ordered…*”

Customer Order Line Many Products appear on many Customer Orders. This is the link table and has a composite primary key made from CustomerOrderId and ProductId which are foreign keys to their respective tables. A quantity columns stores the number of each product ordered. Subtotals are calculated by looking up the price on the product table and multiplying by this number.

|  |  |  |  |
| --- | --- | --- | --- |
| **Stock Item** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Name | Character | Max Length 100, UNIQUE, NOT NULL |  |
| GoodsInQty | Integer | NOT NULL, cannot be less than zero | { 0, 1, 2, …, n } |
| WarehouseQty | Integer | NOT NULL, cannot be less than zero | { 0, 1, 2, …, n } |
| WorkshopQty | Integer | NOT NULL, cannot be less than zero | { 0, 1, 2, …, n } |

Justification from brief:

“*…the individual Stock Items needed to make the Product (for example, a dining room table consists of a table top and four table legs, and a dining room chair consists of a chair back, a chair seat and four chair legs) …*”

Stock Item forms the constituent parts of each product. Some Stock Items are specific to a Product (a table top out of a specific material) while others will be used among many products (joints, screws, fittings). The name of the Stock Item describes it uniquely and includes the unit such as “pack of six,” or “box of 30.” Stock Items will be held as inventory in various quantities at different locations on the premises, such as the Goods Inwards depot, the Warehouse and the Workshop. When Stock Items are running low in all inventory quantities, they need to be re-ordered a Supplier.

|  |  |  |  |
| --- | --- | --- | --- |
| **Product Stock Item** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| ProductId | Integer | Part of composite Primary Key with StockItemId, Foreign Key, NOT NULL | Only existing Product Primary Key Values |
| StockItemId | Integer | Part of composite Primary Key with ProductId, Foreign Key, NOT NULL | Only existing Stock Item Primary Key Values |
| Quantity | Integer | NOT NULL, cannot be less than zero | { 1, 2, …, n } |

Justification from brief:

*(Inferred relation to represent the many to many join between Product and Stock Item)*

“*…the individual Stock Items needed to make the Product (for example, a dining room table consists of a table top and four table legs, and a dining room chair consists of a chair back, a chair seat and four chair legs)…*”

Product Stock Item Many Stock Items are used in many Products. This is the link table and has a composite primary key made from ProductId and StockItemId which are foreign keys to their respective tables. A quantity columns stores the number of each stock item used as a constituent part of a product, e.g. 4 table legs, 1 table top.

|  |  |  |  |
| --- | --- | --- | --- |
| **Supplier** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| CompanyName | Character | UNIQUE, Max Length 50, NOT NULL |  |
| CompanyAddrId | Integer | Foreign Key, NOT NULL | Only existing Address Primary Key Values |
| MainContactId | Integer | Foreign Key, NOT NULL | Only existing CustomerOrder Primary Key Values |

Justification from brief:

“*When standard Stock Items (e.g.table legs, table tops) are out of stock, a Purchase Order is raised, which consists of the supplier’s name, the supplier’s address…*”

Supplier represents an external company that sells Stock Items to BCBF. Each has a unique company name, a correspondence address and a main contact. These latter are stored as Ids to the Address and Person tables respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| **StockItemSupplier** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| StockItemId | Integer | Part of composite Primary Key with SupplierId, Foreign Key, NOT NULL | Only existing Stock Item Primary Key Values |
| SupplierId | Integer | Part of composite Primary Key with StockItemId, Foreign Key, NOT NULL | Only existing Supplier Primary Key Values |
| AgreedPrice | Decimal(8,2) | NOT NULL, cannot be less than zero |  |

Justification from brief: NA – this is an inferred relation to represent the many to many join between Stock Item and Supplier

StockItemSupplier Many suppliers may supply many Stock Items and charge different prices. BCBF will try to order from the one who has agreed the best pricing.

|  |  |  |  |
| --- | --- | --- | --- |
| **PurchaseOrder** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| createdDate | Date | NOT NULL, Default SYSDATE |  |
| completedDate | Date | NULL, cannot be earlier than createdDate |  |
| orderTotal | Decimal(8,2) | NOT NULL, cannot be less than zero | Initially calculated from sum of order line totals. |
| orderStatusId | Integer | Foreign Key, NOT NULL | Only existing OrderStatus Primary Key Values |
| supplierId | Integer | Foreign Key, NOT NULL | Only existing Supplier Primary Key Values |

Justification from brief:

“*When standard Stock Items (e.g. table legs, table tops) are out of stock, a Purchase Order is raised, which consists of…*”

Purchase Order is used to manage the transactions between BCBF and Suppliers of the various Stock Items. Each Purchase order total price calculated at the time the Order is created. Other important information is tracked on the purchase order such as order created date, and order status e.g. Pending, Received as well as the supplier via the foreign key to the Supplier table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Purchase Order Line** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| PurchaseOrderId | Integer | Part of composite Primary Key with StockItemId, Foreign Key, NOT NULL | Only existing PurchaseOrder Primary Key Values |
| StockItemId | Integer | Part of composite Primary Key with PurchaseOrderId, Foreign Key, NOT NULL | Only existing Product Primary Key Values |
| Quantity | Integer | NOT NULL, cannot be less than one | { 1, 2, …, n } |
| PurchasePrice | Decimal(8,2) | NOT NULL, cannot be less than zero |  |

Justification from brief:

*(Inferred from many to many relationship)*

“*… a Purchase Order is raised, which consists of […] the description of the stock items required, the quantity of each stock item required…*”

Purchase Order Line Many Stock Items appear on many Purchase Orders. This is the link table and has a composite primary key made from PurchaseOrderId and StockItemId which are foreign keys to their respective tables. A quantity columns stores the number of each Stock Item ordered. The AgreedPrice is stored here as it may vary from time to time, depending on the supplier, the season and so on. Subtotals are calculated by multiplying the agreed price and the quantity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Cash Transaction** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| Id | Integer | Primary Key, Unique, NOT NULL | { 1, 2, …, n } |
| Amount | Decimal(8,2) | NOT NULL, cannot be less than zero |  |
| createdDate | Date | NOT NULL, Default SYSDATE |  |
| processedDate | Date | NULL, cannot be earlier than createdDate |  |
| completedDate | Date | NULL, cannot be earlier than processedDate |  |

Justification from brief: NA – this is an inferred super-type of other objects.

Cash Transaction is a super type for Receipt (a payment of an Invoiced amount) and Payment (BCBF paying a supplier for purchased Stock Items). Cash Transaction holds the amount of the cash transaction and the transaction created date. The processed date is the date the paperwork is done by the company and the bank date is the date the money enters or leaves the bank account. These could be updated automatically from an accounting system and allow sales execs to have the latest information without needing access to the accounts system itself.

A cash transaction has an XOR relationship with receipts and payments. A cash Transaction must appear either in a Receipt composite Primary Key many or in a Payment composite Primary Key but never both. Furthermore, a cash transaction id may only appear once in either table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Receipt** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| CashTransId | Integer | Part of composite Primary Key with StockItemId, Foreign Key, NOT NULL | Only existing Product Primary Key Values |
| InvoiceId | Integer | Part of composite Primary Key with ProductId, Foreign Key, NOT NULL | Only existing Stock Item Primary Key Values |

Justification from brief:

*(Inferred)*

“*The customer has 30 days from the dispatch date in which to pay the invoice.*”

Receipt is a sub-type of Cash Transaction. A payment pays all or part of an Invoice and is generally entered after the Invoice is created. As a result, an Invoice can have zero to many Receipts.

|  |  |  |  |
| --- | --- | --- | --- |
| **Payment** | | | |
| *Attribute Name* | *Attribute Type* | *Attribute Constraints* | *Attribute Domain* |
| CashTransId | Integer | Part of composite Primary Key with PurchaseOrderId, Foreign Key, NOT NULL | Only existing Product Primary Key Values |
| PurchaseOrderId | Integer | Part of composite Primary Key with CashTransId, Foreign Key, NOT NULL | Only existing CashTransId Primary Key Values |

Justification from brief:

*(Inferred)*

Payment is a sub-type of Cash Transaction. A payment pays all or part of an Purchase Order and is generally entered after the Purchase Order is created. As a result, a Purchase Order can have zero to many Payments.

# Optimisations, Assumptions and Design Choices

Clusters: The following clusters have been defined over sets of tables that are so closely related that a majority of queries will include them together:

|  |  |  |  |
| --- | --- | --- | --- |
| **ClusPerson** | | | |
| *Clustering Key* | *Key Type* | *Cluster Type / Reason* | *Included Tables (Keys)* |
| ClusPersonId | Integer | Index – Transactional data, changing often with every new order. | Person (id)  Customer (personId)  Employee (personId) |

|  |  |  |  |
| --- | --- | --- | --- |
| **ClusCustOrder** | | | |
| *Clustering Key* | *Key Type* | *Cluster Type / Reason* | *Included Tables (Keys)* |
| ClusCustOrderId | Integer | Index – Transactional data, changing often with every new order. | CustomerOrder (id)  WorksOrder (CustomerOrderId)  CustomerOrderLine (CustOrderId) |

|  |  |  |  |
| --- | --- | --- | --- |
| **ClusCashTrans** | | | |
| *Clustering Key* | *Key Type* | *Cluster Type / Reason* | *Included Tables (Keys)* |
| ClusCashTransId | Integer | Index – Transactional data, changing often with every new Cash Transaction. | CashTransaction (id)  Receipt (CashTransId)  Payment (CashTransId) |

|  |  |  |  |
| --- | --- | --- | --- |
| **ClusProduct** | | | |
| *Clustering Key* | *Key Type* | *Cluster Type / Reason* | *Included Tables (Keys)* |
| ClusProductId | Integer | Hash – Static product catalogue data, changing only each season/year when new products are release to market. Equality based queries used when adding Id to new orders | Product (id)  ProductStockItem (ProductId) |

Partitions: Searches on financial data are very likely to be date based, e.g. all invoices between Jan-Mar 2015. The following partitions have been created to reduce the size of such linear searches. It is assumed that the company has a financial audit every 5 years and the Invoices and Payments in previous periods are rarely, if ever, accessed. Arbitrary dates have been selected but these can be changed to align to the company’s financial periods in the final implementation.

Code example for the partitions described above:

CREATE TABLE Invoice (  
id INTEGER NOT NULL  
, createdDate DATE DEFAULT SYSDATE NOT NULL  
, dueDate DATE DEFAULT SYSDATE + 30 NOT NULL  
, worksOrderId INTEGER NOT NULL  
, CONSTRAINT pk\_\_invoice PRIMARY KEY (id)  
, CONSTRAINT fk\_\_invoice\_workorder FOREIGN KEY (worksOrderId) REFERENCES WorksOrder (id)  
/\* Adding a unique index on worksOrderId enforces   
the one to one relationship with WorksOrder  
and also speeds up queries joining the two tables  
which is essential as it will happen all the time \*/  
, CONSTRAINT uq\_\_invoice\_workorder UNIQUE (worksOrderId)  
) /\* Archive old Invoices every 5 years because they're no longer accessed after an audit \*/  
PARTITION BY RANGE (createdDate)  
 (PARTITION Inv\_Archive2010 VALUES LESS THAN ('1-Jan-2010')  
 , PARTITION Inv\_Archive2015 VALUES LESS THAN ('1-Jan-2015')  
 , PARTITION Inv\_Archive2020 VALUES LESS THAN ('1-Jan-2020'));  
  
/\* Index on FK to WorksOrder is desirable as this table will often be joined to WO for queries /\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

-----

CREATE TABLE PurchaseOrder (  
id INTEGER NOT NULL  
, createdDate DATE DEFAULT SYSDATE NOT NULL  
, supplierId INTEGER NOT NULL  
, orderTotal NUMBER(5) NOT NULL  
, orderStatusId INTEGER NOT NULL  
, completedDate DATE DEFAULT SYSDATE NULL  
, CONSTRAINT pk\_\_purchorder PRIMARY KEY (id)  
, CONSTRAINT fk\_\_purchord\_supp FOREIGN KEY (supplierId) REFERENCES Supplier (id)  
, CONSTRAINT fk\_\_purchorder\_status FOREIGN KEY (orderStatusId) REFERENCES OrderStatus (id)  
/\* Avoid orders with zero or negative values \*/  
, CONSTRAINT ck\_\_worksorder\_total CHECK (orderTotal >= 0.0)  
/\* Can't complete earlier than the order was created! \*/  
, CONSTRAINT ck\_\_worksorder\_completdate CHECK (completedDate > createdDate)  
) /\* Archive old POs every 5 years because they're no longer accessed after an audit \*/  
PARTITION BY RANGE (createdDate)  
 (PARTITION PO\_Archive2010 VALUES LESS THAN ('1-Jan-2010')  
 , PARTITION PO\_Archive2015 VALUES LESS THAN ('1-Jan-2015')  
 , PARTITION PO\_Archive2020 VALUES LESS THAN ('1-Jan-2020'));  
/\* no index on order status because that is a tiny table of just a few category names  
which is faster to search linearly \*/

/\* index on FK to Supplier table because join to supplier table will be very common \*/  
CREATE INDEX idx\_\_purchord\_supp ON PurchaseOrder (supplierId);

Nested Tables: As discussed in the section on the Address table, Nested Tables were considered useful for the purpose of managing address data. However, it proved over complicated to apply the required contraints for Non-nullable values such as Address Line 1, townOrCity and PostCode. No other opportunity to use Nested Tables was found in the implementation.

# Security, Views, Roles and Privileges

Role Based Security: Industry best practices suggest that role based security is essential both for maintainability and for upholding high security standards in the long term. Role-based security means that individual users are never granted permission to database objects. They are instead assigned to one or more Roles which have been assigned Role based permissions to database objects. The benefit of this approach is that it is easy to keep track of where permissions have been applied, and changes are easy to apply in bulk to all affected users by simply updating the roles.

The following Roles have been identified for the initial implementation, although more can be added at any time:

Sales\_Exec: Can create, view and edit Customers and Customer Orders

Sales\_Admin: Can change the Person table, in case any previously unforeseen names or titles come up and contraints need to be changed. Whoever is granted this role must be highly trusted and well trained.

Production\_Exec: Can view a listing of Works Orders to plan the day’s work in the workshop.

Inventory Manager: Can view a listing of stock items which shows how many are in stock of each item and how many are needed for forward orders.

Views: The following code snippets implement the Views required for the above Security arrangements and then assign the Roles as described.

/\* Users, Roles, Views and Security \*/  
CREATE VIEW v\_CustomerDetails AS   
SELECT c.id AS "Customer No."  
, p.givenName AS "Given Name"  
, p.familyName AS "Family Name"  
, p.dateOfBirth AS "Date of Birth"  
, p.email AS "Email Address"  
FROM Person p  
INNER JOIN Customer c  
 ON p.id = c.personId;

CREATE VIEW v\_CustomerOrderDetails AS  
SELECT c.id AS "Customer No."  
, p.givenName AS "Given Name"  
, p.familyName AS "Family Name"  
, p.email AS "Email Address"  
, co.createdDate AS "Order Date"  
, co.shipDate AS "Ship Date"  
, co.orderTotal AS "Order Total"  
, os.statusName AS "Order Status"  
FROM Person p  
INNER JOIN Customer c  
 ON p.id = c.personId  
INNER JOIN CustomerOrder co  
 ON c.id = co.customerId  
INNER JOIN OrderStatus os  
 ON co.orderStatusId = os.id;

CREATE VIEW v\_WorksOrdersListing AS  
SELECT wo.id AS "Works Order No."  
, co.id AS "Customer Order No."  
, co.customerId AS "Customer No."  
, wo.createdDate AS "Order Date"  
, wo.requiredDate AS "Required Date"  
, p.givenName || ' ' || p.familyName AS "Assigned To"  
, wo.completedDate AS "Completed Date"  
, p2.givenName || ' ' || p2.familyName AS "Completed By"  
FROM WorksOrder wo  
INNER JOIN CustomerOrder co  
 ON wo.customerOrderId = co.id  
INNER JOIN Employee e  
 ON wo.assignedToId = e.id  
INNER JOIN Person p  
 ON e.personId = p.id  
LEFT OUTER JOIN Employee e2  
 ON wo.completedById = e2.id  
LEFT OUTER JOIN Person p2  
 ON e2.personId = p2.id;  
   
CREATE VIEW v\_InventoryOverview AS  
SELECT si.id AS "SKU No."  
, si.name "Item Description"  
, SUM(si.goodsInQty + si.warehouseQty + si.workshopQty) AS "Inventory Quantity"  
, nvl(foq.ForwardOrdersQuantity, 0) AS "Forward Orders"  
FROM StockItem si  
LEFT OUTER JOIN (  
 SELECT psi.stockItemId  
 , col.quantity \* psi.quantity AS ForwardOrdersQuantity  
 FROM ProductStockItem psi  
 INNER JOIN Product p  
  ON psi.productId = p.id  
 INNER JOIN CustomerOrderLine col  
  ON p.id = col.productId  
 INNER JOIN CustomerOrder co  
  ON col.customerOrderId = co.id  
 INNER JOIN OrderStatus os  
  ON co.orderStatusId = os.id  
 WHERE os.statusName = 'Pending'  
 ) foq  
 ON si.id = foq.stockItemId  
GROUP BY si.id  
, si.name  
, si.goodsInQty  
, si.warehouseQty  
, si.workshopQty  
, foq.ForwardOrdersQuantity;

CREATE PROFILE bcbf\_default LIMIT  
FAILED\_LOGIN\_ATTEMPTS 5  
PASSWORD\_LIFE\_TIME 60  
PASSWORD\_REUSE\_TIME 60  
PASSWORD\_REUSE\_MAX 5  
PASSWORD\_LOCK\_TIME 1/24  
PASSWORD\_GRACE\_TIME 10;

CREATE ROLE Role\_Sales\_Exec IDENTIFIED EXTERNALLY;  
GRANT SELECT ON v\_CustomerDetails TO Role\_Sales\_Exec;  
GRANT SELECT ON v\_CustomerOrderDetails TO Role\_Sales\_Exec;

CREATE ROLE Role\_Sales\_Admin IDENTIFIED EXTERNALLY;  
GRANT ALTER ON Person TO Role\_Sales\_Admin;

CREATE ROLE Role\_Production\_Exec IDENTIFIED EXTERNALLY;  
GRANT SELECT ON v\_WorksOrdersListing TO Role\_Sales\_Exec;

CREATE ROLE Role\_Inventory\_Manager IDENTIFIED EXTERNALLY;  
GRANT SELECT ON v\_InventoryOverview TO Role\_Sales\_Exec;

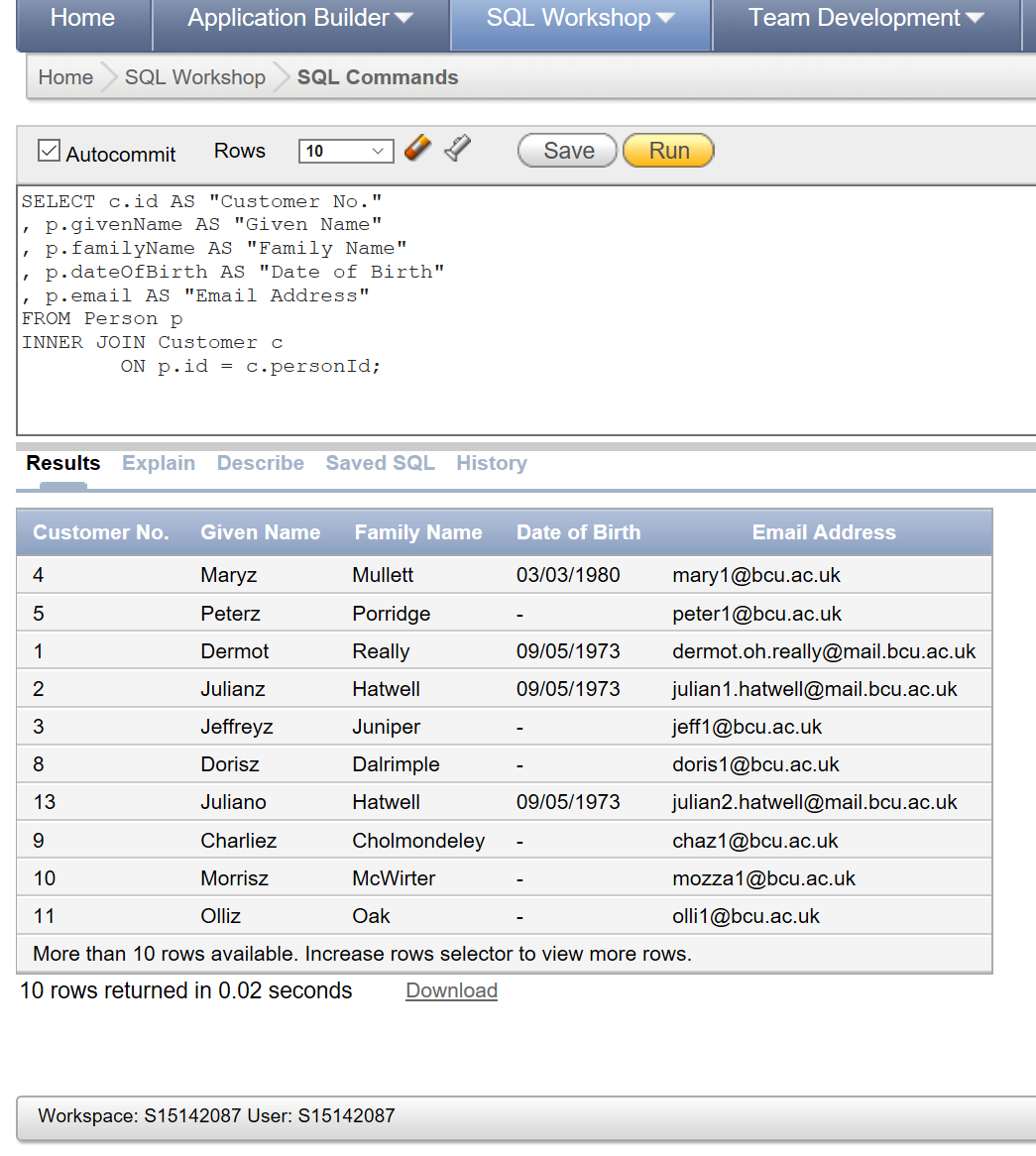
CREATE USER Bob\_Downs  
IDENTIFIED BY AZ7BC2  
PROFILE bcbf\_default;  
GRANT CREATE SESSION, Role\_Sales\_Exec, Role\_Sales\_Admin TO Bob\_Downs;

CREATE USER Vic\_Murphy  
IDENTIFIED BY TigerLilly  
PROFILE bcbf\_default;  
GRANT CREATE SESSION, Role\_Production\_Exec TO Vic\_Murphy;

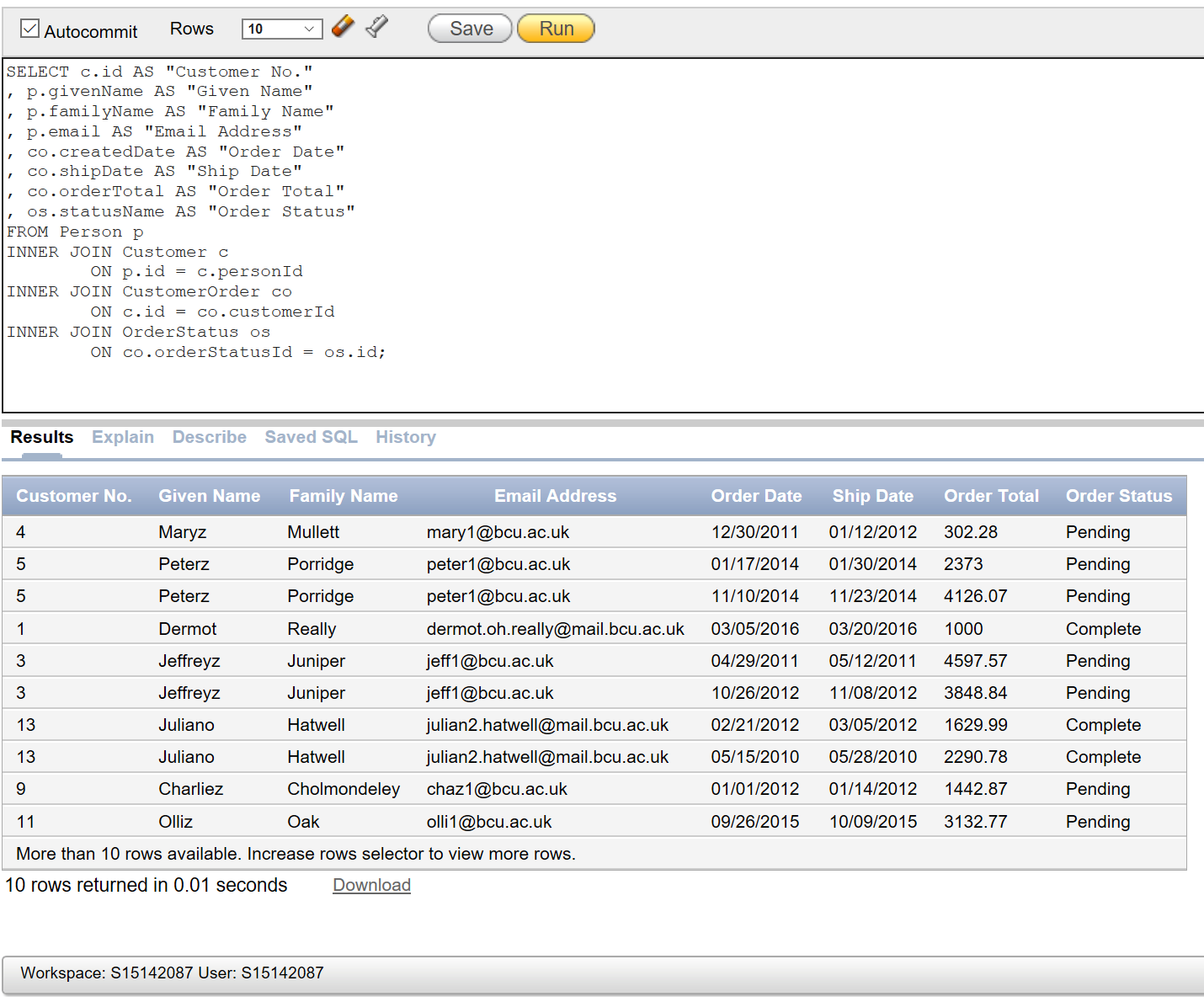
CREATE USER Laurie\_Mapplethorpe  
IDENTIFIED BY theShznzz  
PROFILE bcbf\_default;  
GRANT CREATE SESSION, Role\_Inventory\_Manager TO Laurie\_Mapplethorpe;

# Appendix A: Screenshots of realistic sample queries and the results (next page).

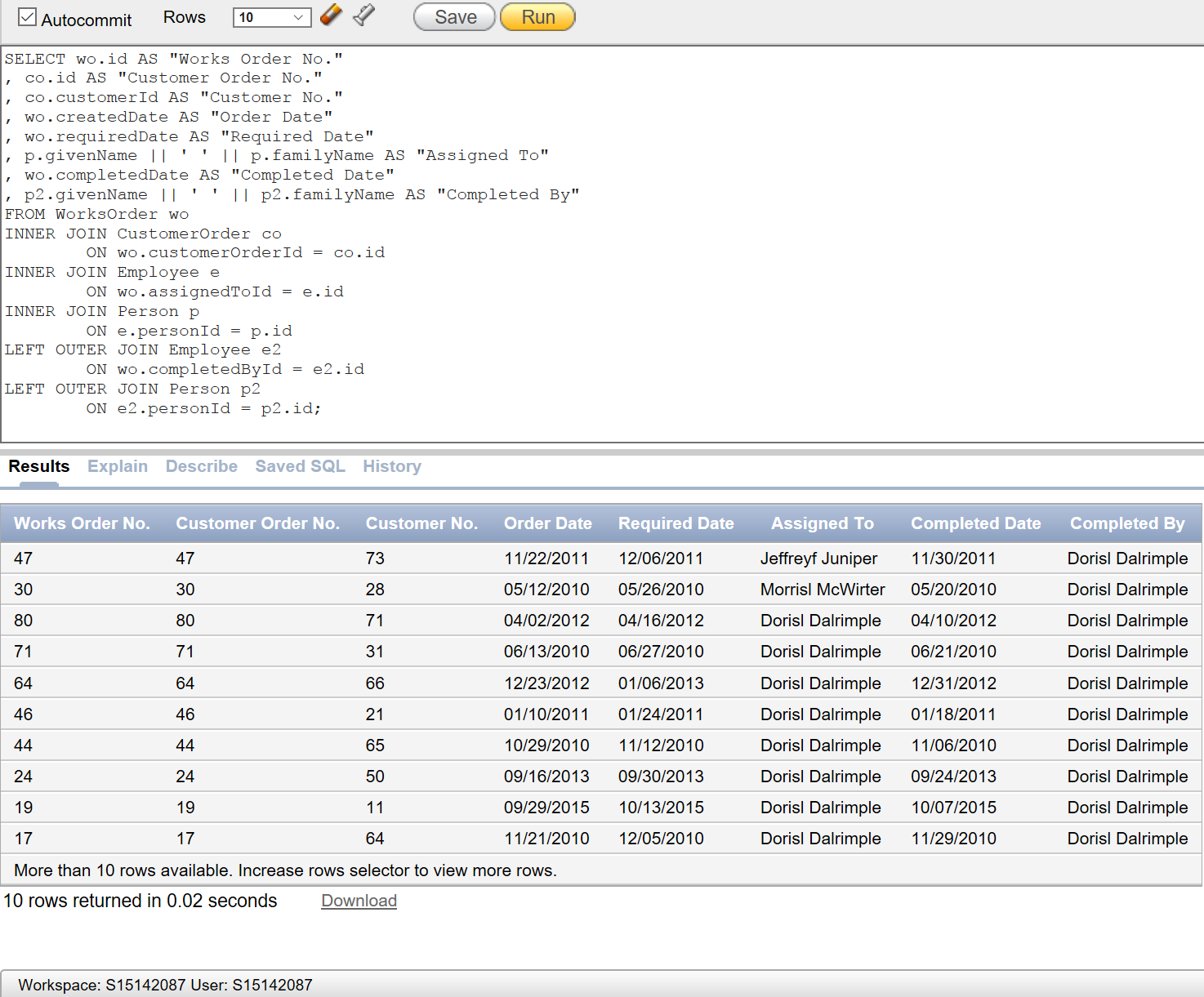
From the Customer details View:



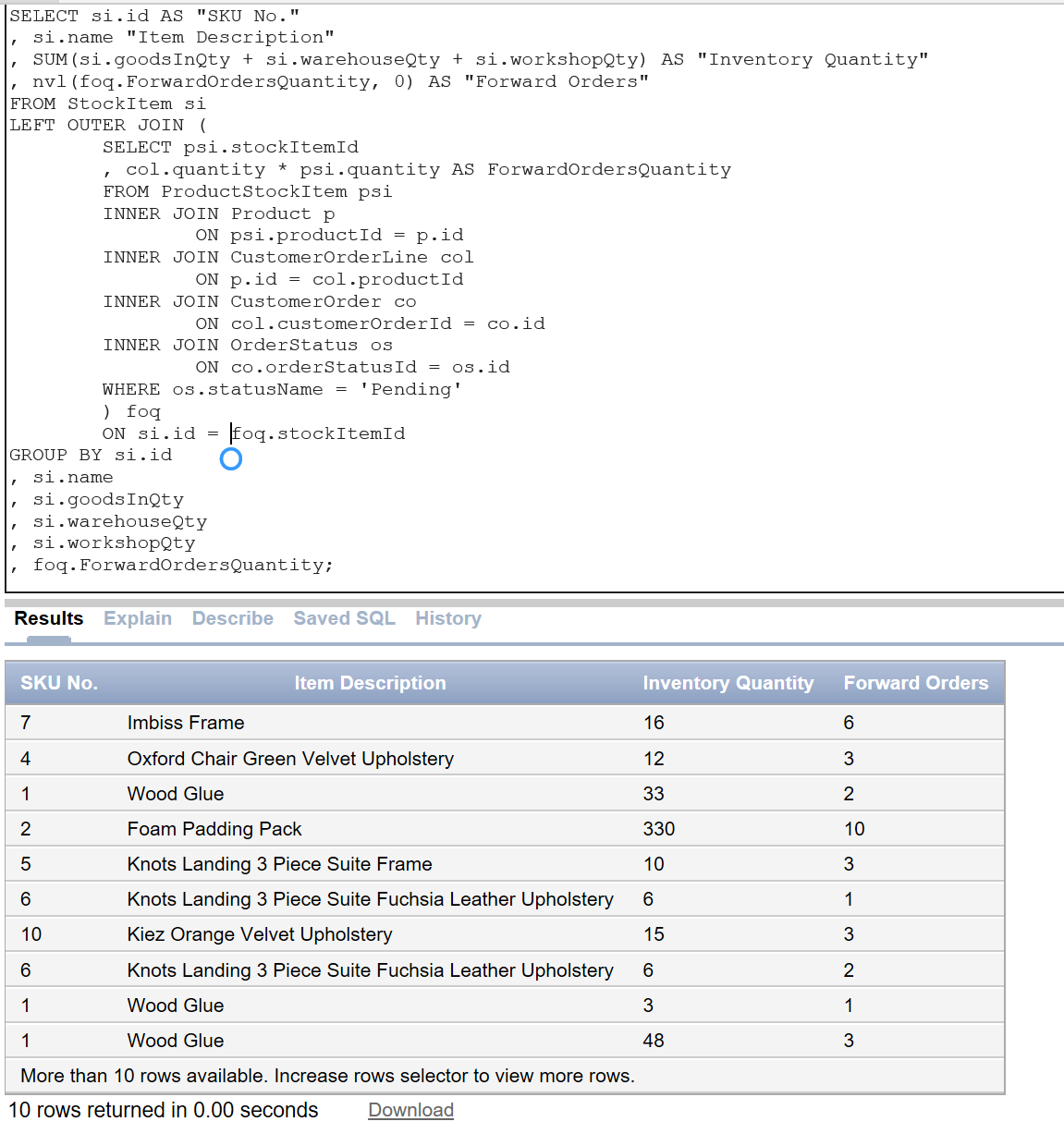
From the Customer Order Details view:



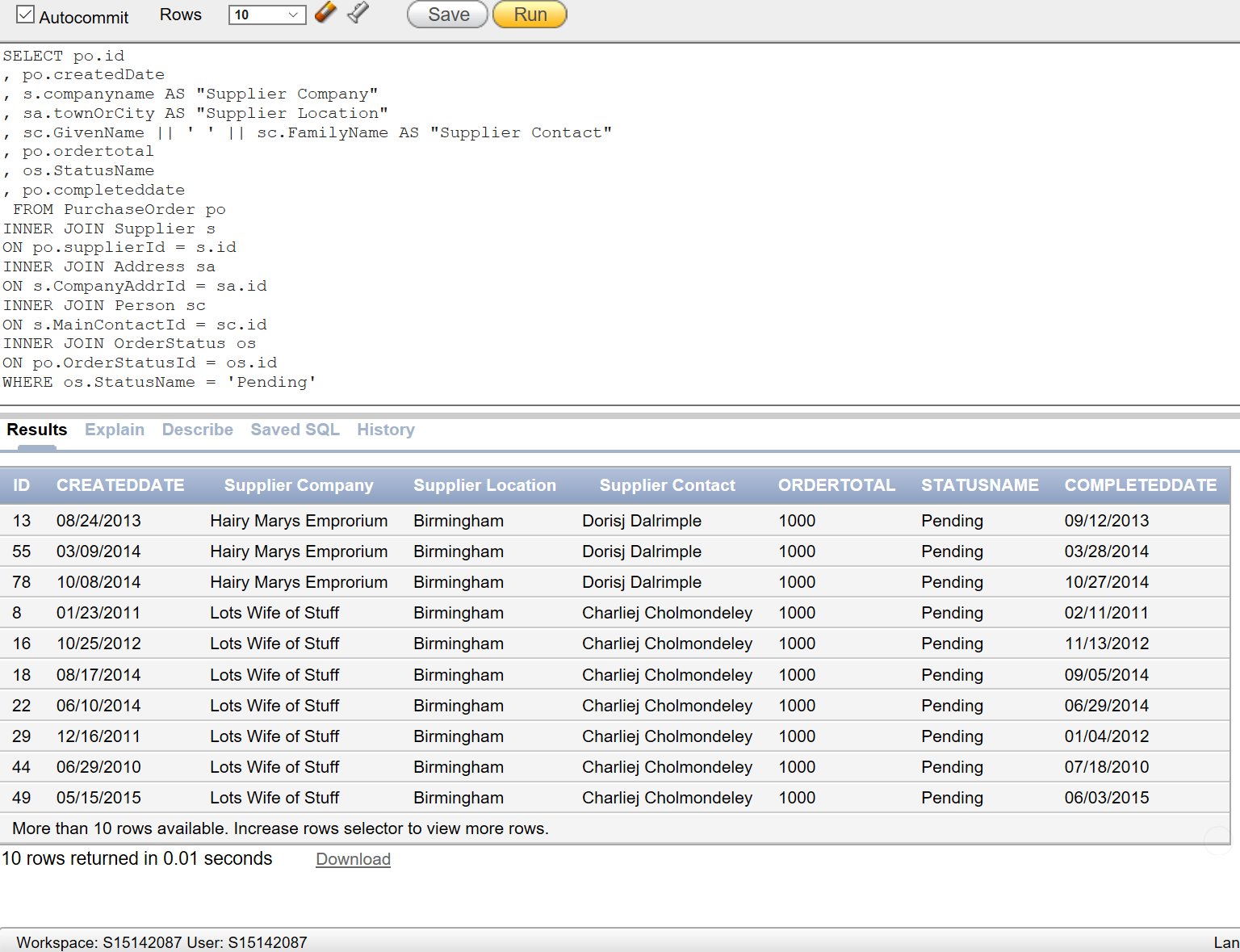
From the WorksOrderListing view:



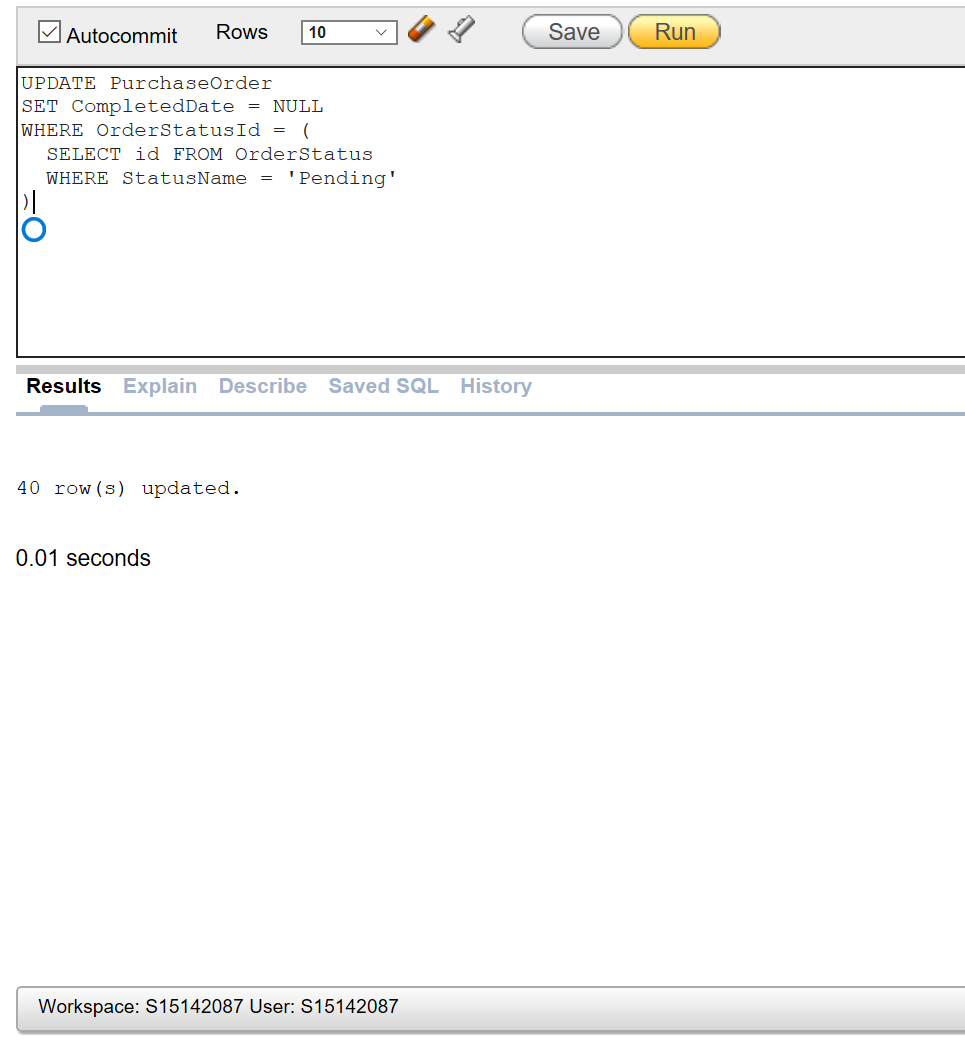
From the Inventory Overview view:



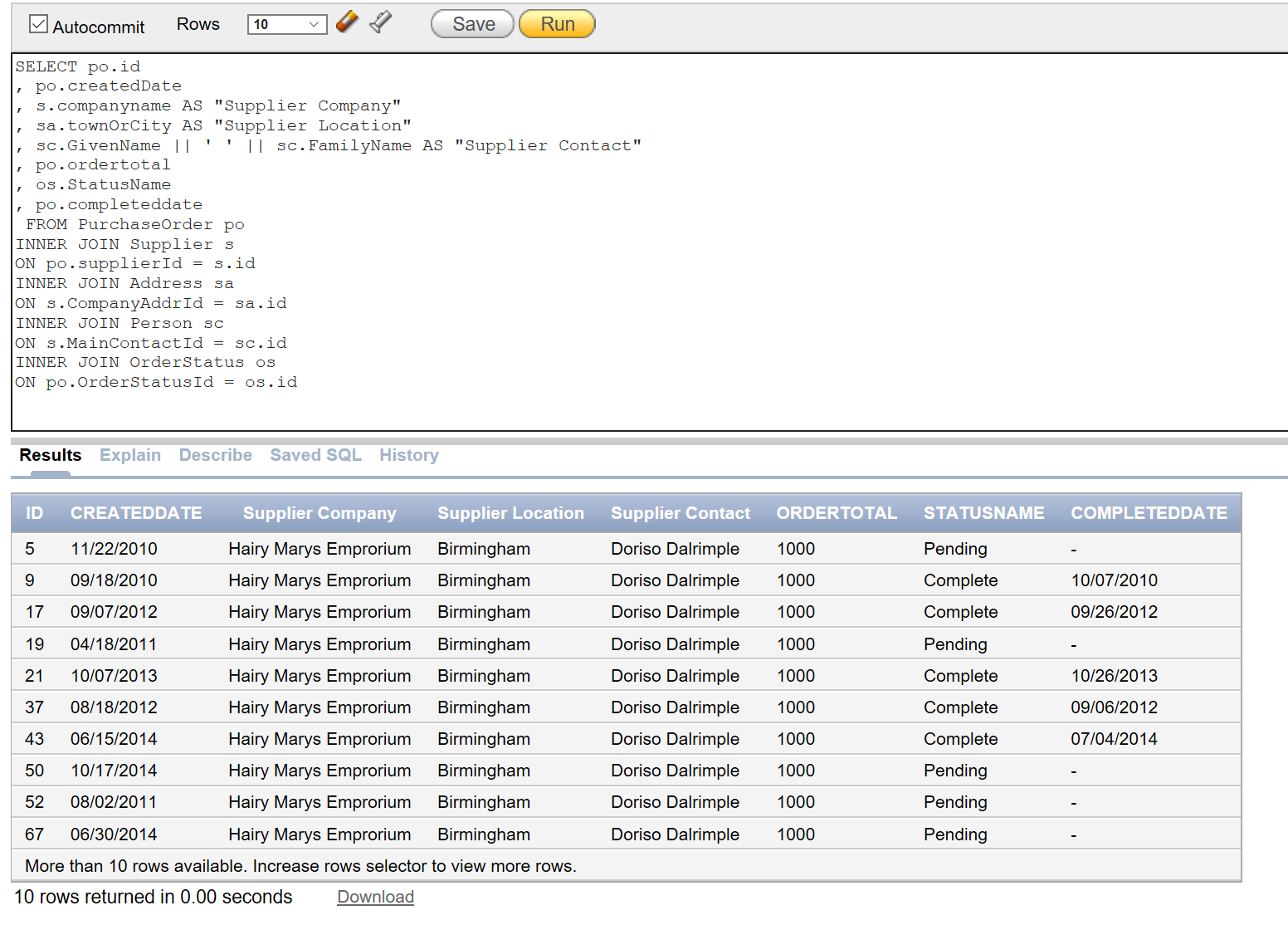
Following the creation of demonstration data, there are a number of invalid data entries. For example, if a PurchaseOrder is pending, it should have a NULL completed date. Currently this is not the case:



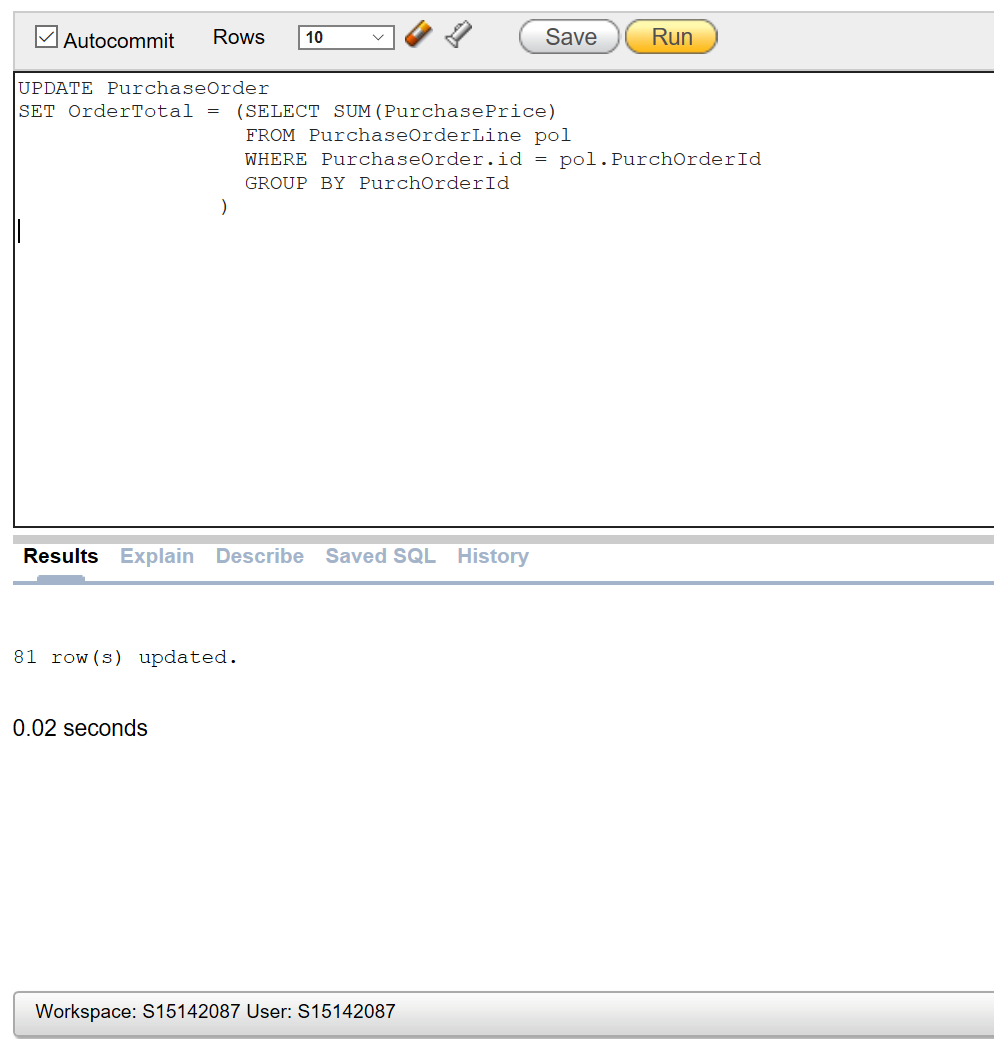
This can be tidied up with the following UPDATE statement:



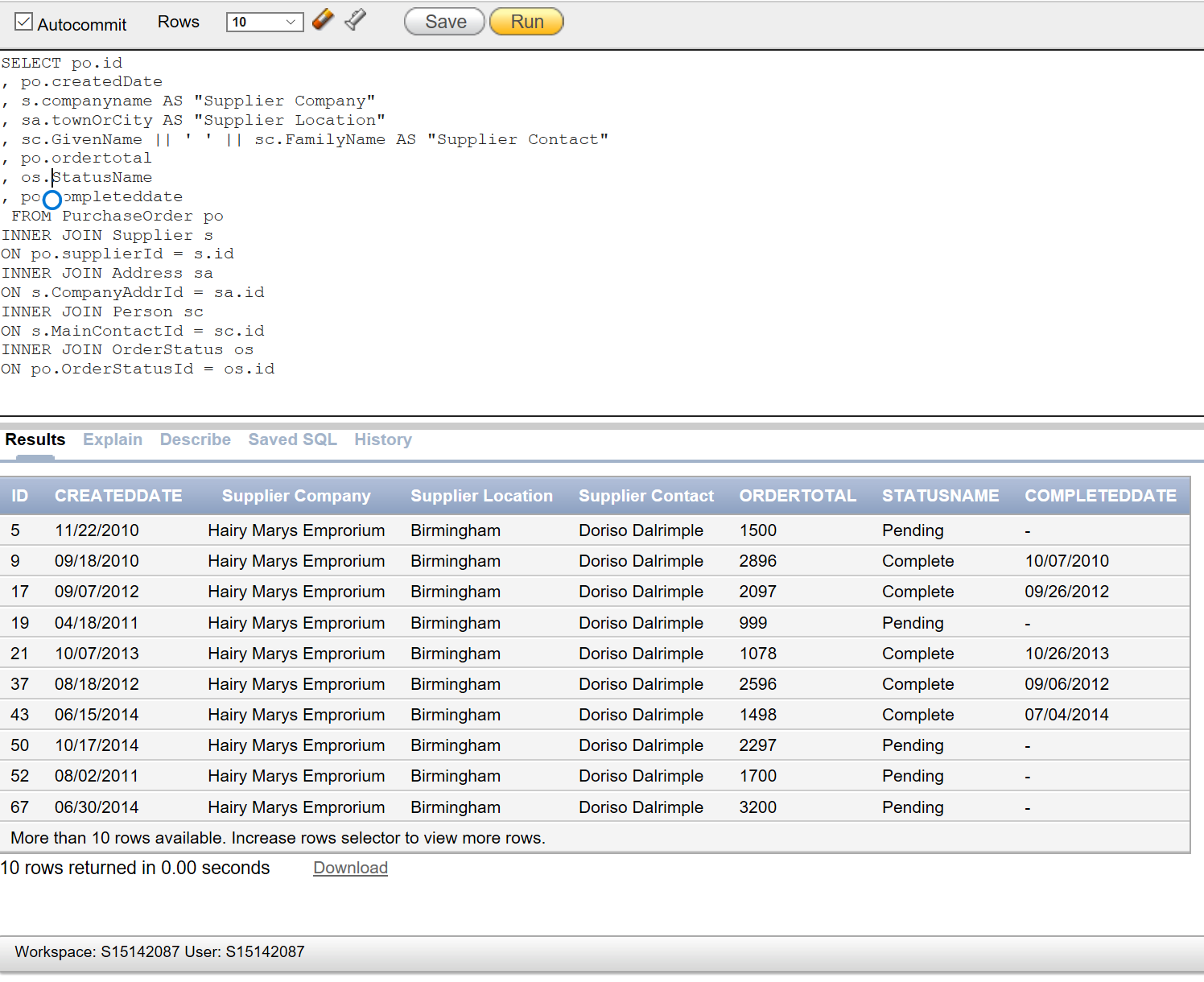
Re-run the PurchaseOrder query:



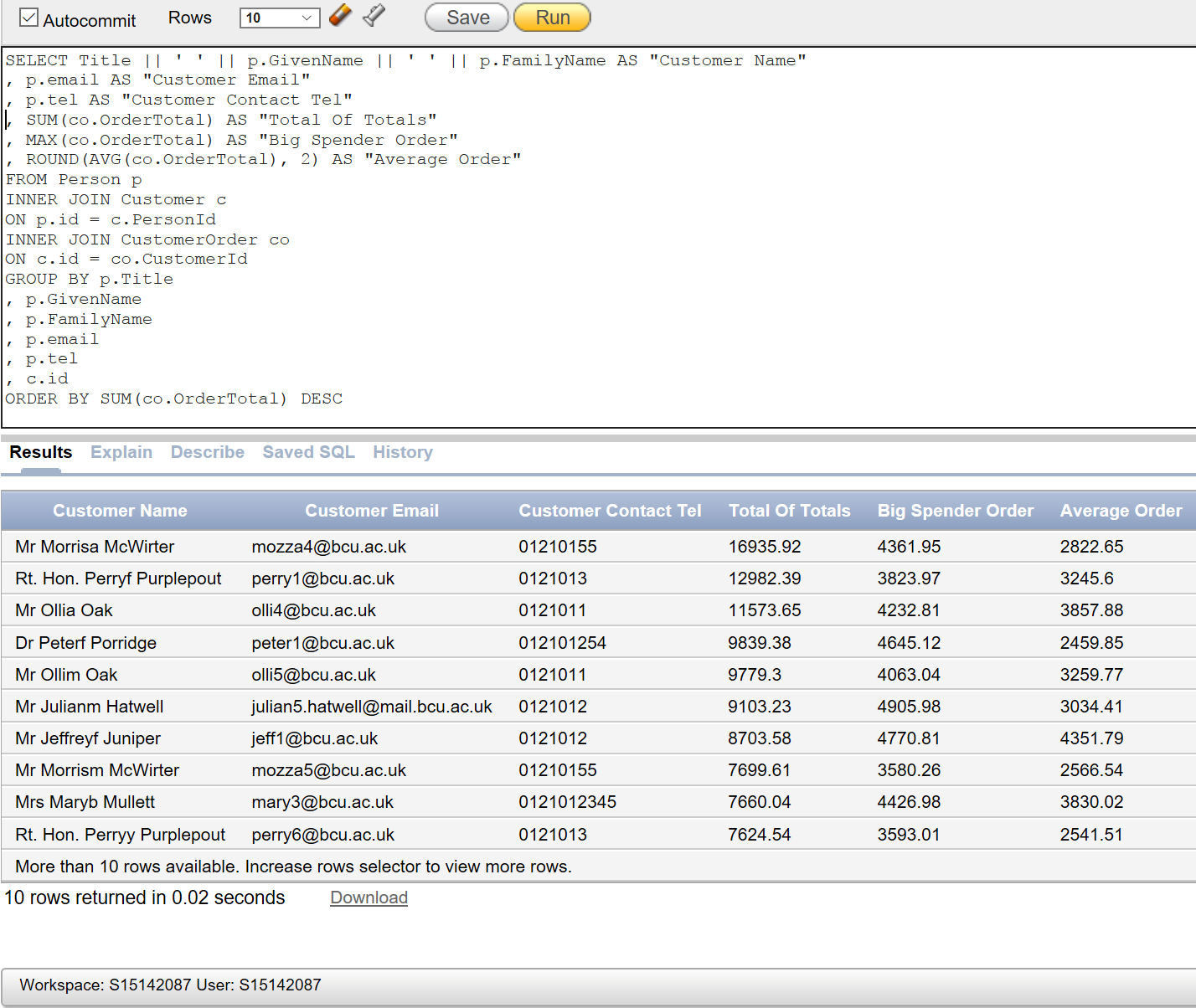
Another problem with the demo data is that the order totals were inserted before the order lines, and so the two don’t relate which they would in the real life setting. This can be fixed with the following update script:



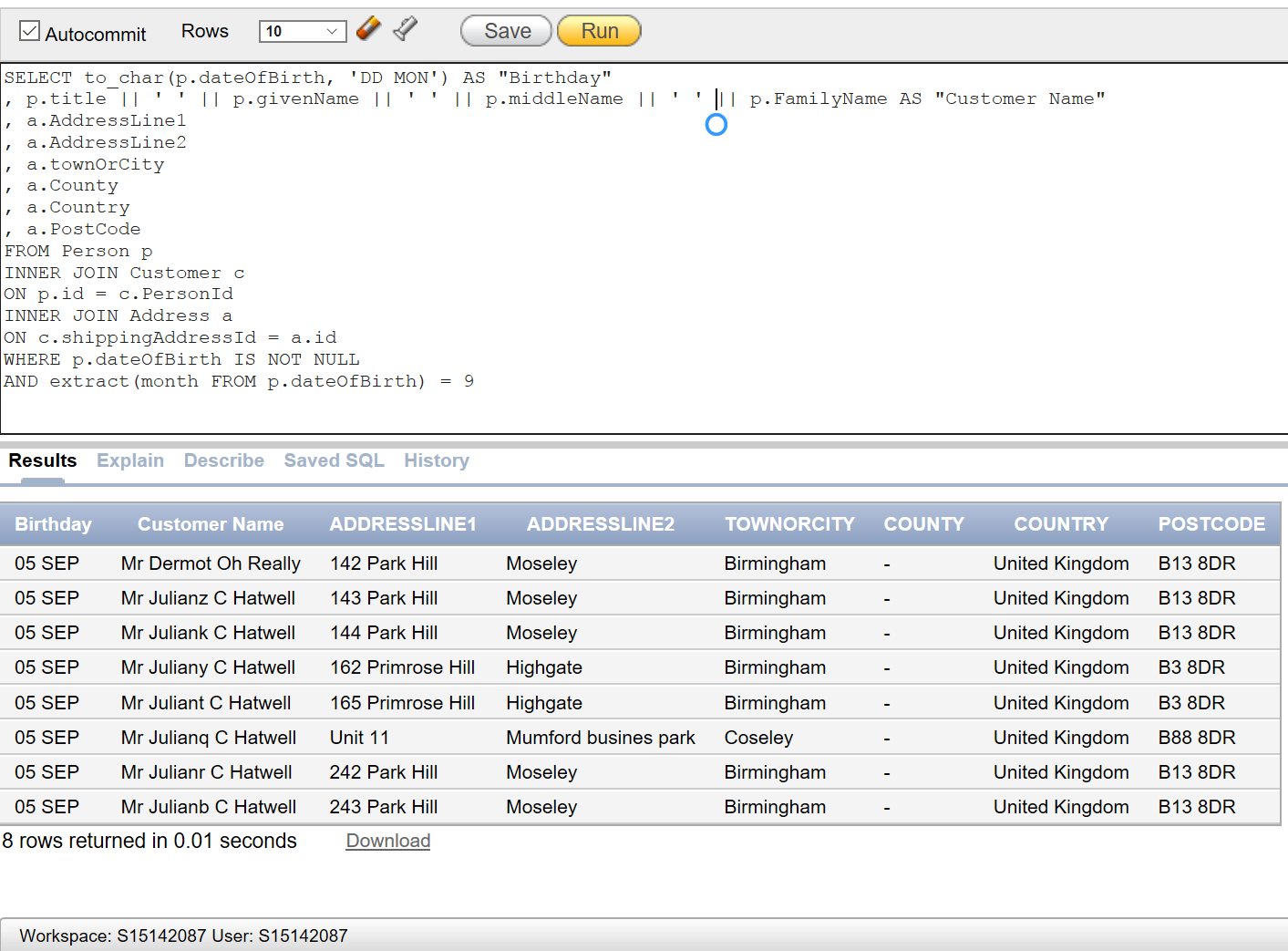
Rerunning the original Purchase Order query:



The business may want to look at their most valuable customers:



The marketing department want to identify all the people with a birthday in September to send out a promotional birthday card. This uses an index on Person.dateOfBirth to speed up the query:



# Appendix B: Script for creating and populating the data database with objects described and sample data:

DROP USER Bob\_Downs CASCADE;

DROP USER Vic\_Murphy CASCADE;

DROP USER Laurie\_Mapplethorpe CASCADE;

DROP ROLE Role\_Sales\_Exec;

DROP ROLE Role\_Sales\_Admin;

DROP ROLE Role\_Production\_Exec;

DROP ROLE Role\_Inventory\_Manager;

DROP PROFILE bcbf\_default;

DROP VIEW v\_InventoryOverview;

DROP VIEW v\_WorksOrdersListing;

DROP VIEW v\_CustomerOrderDetails;

DROP VIEW v\_CustomerDetails;

DROP TABLE Payment;

DROP TABLE Receipt;

DROP TABLE CashTransaction;

DROP TABLE PurchaseOrderLine;

DROP TABLE PurchaseOrder;

DROP TABLE StockItemSupplier;

DROP TABLE Supplier;

DROP TABLE ProductStockItem;

DROP TABLE StockItem;

DROP TABLE CustomerOrderLine;

DROP TABLE Product;

DROP TABLE Invoice;

DROP TABLE WorksOrder;

DROP TABLE CustomerOrder;

DROP TABLE CustOrderAddress;

DROP TABLE OrderStatus;

DROP TABLE Employee;

DROP TABLE Customer;

DROP TABLE Address;

DROP TABLE Person;

DROP CLUSTER ClusProduct;

DROP CLUSTER ClusCashTrans;

DROP CLUSTER ClusCustOrder;

DROP CLUSTER ClusPerson;

/\* all tables that "descend" from Person will be stored in a cluster

Person - Customer - Employee

because they will almost always be searched and joined together.

Index is cluster chosen. Although the data is changing very slowly, it is still changing.

If the company grew, there would be the risk of having to rebuild a hash cluster.

New customers at added at most a couple of time per day.

New employees are added at most once or twice per month. \*/

CREATE CLUSTER ClusPerson (clusPersonId INTEGER);

CREATE INDEX idx\_\_clusperson ON CLUSTER ClusPerson;

/\* all tables that "descend" from CustomerOrder will be stored in a cluster

CustomerOrder - WorksOrder - CustomerOrderLine

because they will almost always be searched and joined together.

Index is cluster chosen. This is the bread and butter operation of the database

and probably the fastest moving. \*/

CREATE CLUSTER ClusCustOrder (clusCustOrderId INTEGER);

CREATE INDEX idx\_\_clusCustOrder ON CLUSTER ClusCustOrder;

/\* all tables that "descend" from CashTransaction will be stored in a cluster

CashTransaction - Receipt - Payment

because they will almost always be searched and joined together.

Index is cluster chosen. This another fast(ish) moving set of operational data. \*/

CREATE CLUSTER ClusCashTrans (clusCashTransId INTEGER);

CREATE INDEX idx\_\_clusCashTrans ON CLUSTER ClusCashTrans;

/\* Product and ProductStockItem will be clustered.

They will frequently be searched together using Product.Id = ProductStockItem.ProductId

This data will change infrequently, perhaps per season or annually

And the size of the product catalogue will not grow linearly with cutomers and orders \*/

CREATE CLUSTER ClusProduct (ClusProductId INTEGER)

SIZE 1305

HASH IS ClusProductId HASHKEYS 200;

/\* Size is (255 + 6) \* 5 (integer, 2-precision number and varchar(100) + varchar(250) from Product table

plus three integers from ProductStockItem table) multiplied by average of 5 stock items per Product

Maximum 200 products at any time in the catalogue \*/

CREATE TABLE Person (

id INTEGER NOT NULL

, title VARCHAR(10) NULL

, givenName VARCHAR2(20) NOT NULL

, middleName VARCHAR2(20) NULL

, familyName VARCHAR2(20) NOT NULL

, dateOfBirth DATE NULL

, email VARCHAR2(50) NOT NULL

, tel VARCHAR2(20) NULL

, CONSTRAINT pk\_\_person PRIMARY KEY (id)

, CONSTRAINT uq\_\_pers\_email UNIQUE (email)

, CONSTRAINT chk\_\_title\_value CHECK (title IN ('Mr', 'Ms', 'Mrs', 'Miss', 'Dr', 'Sir', 'Rt. Hon.', 'Lord', 'HRH'))

, CONSTRAINT chk\_\_gname\_format CHECK (REGEXP\_INSTR(givenName, '[0-9!"£$€&\*()\_=+<>,\/]', 1, 1) = 0)

, CONSTRAINT chk\_\_mname\_format CHECK (REGEXP\_INSTR(givenName, '[0-9!"£$€&\*()\_=+<>,\/]', 1, 1) = 0)

, CONSTRAINT chk\_\_fname\_format CHECK (REGEXP\_INSTR(givenName, '[0-9!"£$€&\*()\_=+<>,\/]', 1, 1) = 0)

, CONSTRAINT chk\_\_email\_format CHECK (REGEXP\_LIKE(email, '[a-zA-Z0-9.\_%-]+@[a-zA-Z0-9.\_%-]+\.[a-zA-Z]{2,4}'))

)

CLUSTER ClusPerson (Id);

-- For marketing purposes, send out a card on customers birthdays

-- Adding an index of Date of Birth will allow efficient linear searches

-- of up and coming birthdays

CREATE INDEX idx\_\_pers\_dob ON Person(dateOfBirth);

-- Person Data

-- first row

INSERT INTO Person VALUES (1, 'Mr', 'Dermot', 'Oh' , 'Really', '5-Sep-1973', 'dermot.oh.really@mail.bcu.ac.uk', '0121012');

DECLARE max\_value NUMBER := 10;

randchr CHAR(1);

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

-- create a random character

SELECT DBMS\_RANDOM.STRING('L', 1) INTO randchr FROM Dual;

--insert the values in to the table created above

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Mr', 'Julian' || randchr, 'C' , 'Hatwell', '5-Sep-1973', 'julian' || to\_char(i) || '.hatwell@mail.bcu.ac.uk', '0121012');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Mr', 'Jeffrey' || randchr, 'J.', 'Juniper', NULL, 'jeff' || to\_char(i) || '@bcu.ac.uk', '0121012');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Mrs', 'Mary' || randchr, NULL, 'Mullett', '3-Mar-1980', 'mary' || to\_char(i) || '@bcu.ac.uk', '0121012345');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Dr', 'Peter' || randchr, NULL, 'Porridge', NULL, 'peter' || to\_char(i) || '@bcu.ac.uk', '012101254');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Ms', 'Sally' || randchr, NULL, 'Cinammon', NULL, 'Sally' || to\_char(i) || '@bcu.ac.uk', '0121012243');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Ms', 'Phenella' || randchr, NULL, 'Phennel', NULL, 'phen' || to\_char(i) || '@bcu.ac.uk', '012101245');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Ms', 'Doris' || randchr, 'Dalia', 'Dalrimple', NULL, 'doris' || to\_char(i) || '@bcu.ac.uk', '0121012765');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Sir', 'Charlie' || randchr, NULL, 'Cholmondeley', NULL, 'chaz' || to\_char(i) || '@bcu.ac.uk', '01210123');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Mr', 'Morris' || randchr, NULL, 'McWirter', NULL, 'mozza' || to\_char(i) || '@bcu.ac.uk', '01210155');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Mr', 'Olli' || randchr, 'Oscar', 'Oak', NULL, 'olli' || to\_char(i) || '@bcu.ac.uk', '0121011');

INSERT INTO Person VALUES ((SELECT MAX(id) + 1 FROM Person)

, 'Rt. Hon.', 'Perry' || randchr, NULL, 'Purplepout', NULL, 'perry' || to\_char(i) || '@bcu.ac.uk', '0121013');

--End of loop

END LOOP;

-- end of script

END;

/

COMMIT;

CREATE TABLE Address (

id INTEGER NOT NULL

, addressLine1 VARCHAR2(50) NOT NULL

, addressLine2 VARCHAR2(50) NULL

, townOrCity VARCHAR2(20) NOT NULL

, county VARCHAR2(20) NULL

, country VARCHAR2(20) DEFAULT ('United Kingdom') NOT NULL

, postCode CHAR(10) NULL

, CONSTRAINT pk\_\_address PRIMARY KEY (id)

);

-- Data

-- first row

INSERT INTO Address VALUES (1, '41 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

DECLARE max\_value NUMBER := 10;

randchr CHAR(1);

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

--insert the values in to the table created above

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '42 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '43 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '44 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '60 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '61 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '62 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '63 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '64 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), to\_char(i) || '65 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), 'Unit 1' || to\_char(i), 'Mumford busines park', 'Coseley', NULL, 'United Kingdom', 'B88 8DR');

INSERT INTO Address VALUES ((SELECT MAX(id) + 1 FROM Address), 'Unit 2' || to\_char(i), 'Mumford busines park', 'Coseley', NULL, 'United Kingdom', 'B88 8DR');

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Customer (

id INTEGER NOT NULL

, billingAddressId INTEGER NOT NULL

, shippingAddressId INTEGER NOT NULL

, personId INTEGER NOT NULL

, CONSTRAINT pk\_\_customer PRIMARY KEY (id)

, CONSTRAINT fk\_\_cust\_billaddr FOREIGN KEY (billingAddressId) REFERENCES Address (id)

, CONSTRAINT fk\_\_cust\_shipaddr FOREIGN KEY (shippingAddressId) REFERENCES Address (id)

, CONSTRAINT fk\_\_cust\_pers FOREIGN KEY (personId) REFERENCES Person (id)

, CONSTRAINT uq\_\_cust\_pers UNIQUE (personId)

) /\* adding to ClusPerson as described above \*/

CLUSTER ClusPerson (personId);

/\* Adding a unique index on personId enforces

the one to one relationship with person

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

/\* Index is desirable as this table will always be joined to person for queries

/\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

-- Data

-- first row

INSERT INTO Customer VALUES (1, 1, 2, 1);

DECLARE max\_value NUMBER := 8;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 3 + (i - 1), 3 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 4 + (i - 1), 4 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 5 + (i - 1), 6 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 7 + (i - 1), 7 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 1 + (i - 1), 2 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 3 + (i - 1), 3 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 8 + (i - 1), 8 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 1 + (i - 1), 4 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

INSERT INTO Customer VALUES ((SELECT MAX(id) + 1 FROM Customer), 8 + (i - 1), 9 + (i - 1), (SELECT MAX(personId) + 1 FROM Customer));

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Employee (

id INTEGER NOT NULL

, homeAddressId INTEGER NOT NULL

, personId INTEGER NOT NULL

, CONSTRAINT pk\_\_employee PRIMARY KEY (id)

, CONSTRAINT fk\_\_cust\_homeaddr FOREIGN KEY (homeAddressId) REFERENCES Address (id)

, CONSTRAINT fk\_\_emp\_pers FOREIGN KEY (personId) REFERENCES Person (id)

, CONSTRAINT uq\_\_emp\_pers UNIQUE (personId)

) /\* adding to ClusPerson as described above \*/

CLUSTER ClusPerson (personId);

/\* Adding a unique index on personId enforces

the one to one relationship with person

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

/\* Index is desirable as this table will always be joined to person for queries

UQ Constraint already created an index - tested by trying to create an explicit index. \*/

-- Data

-- First Row

INSERT INTO Employee VALUES (1, (SELECT MAX(billingAddressId) + 1 FROM Customer), (SELECT MAX(personId) + 1 FROM Customer));

COMMIT;

DECLARE max\_value NUMBER := 10;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 2..max\_value LOOP

INSERT INTO Employee VALUES (i, (SELECT MAX(homeAddressId) + 1 FROM Employee), (SELECT MAX(personId) + 1 FROM Employee));

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE OrderStatus (

id INTEGER NOT NULL

, statusName CHAR(10)

, CONSTRAINT pk\_\_orderStatus PRIMARY KEY (id)

/\* Mustn't have two names the same or it would get messy \*/

, CONSTRAINT uq\_\_orderStatusName UNIQUE (statusName));

INSERT INTO OrderStatus VALUES (1, 'Pending');

INSERT INTO OrderStatus VALUES ((SELECT MAX(id) + 1 FROM OrderStatus), 'Complete');

INSERT INTO OrderStatus VALUES ((SELECT MAX(id) + 1 FROM OrderStatus), 'Despatched');

INSERT INTO OrderStatus VALUES ((SELECT MAX(id) + 1 FROM OrderStatus), 'Paid');

COMMIT;

CREATE TABLE CustOrderAddress (

id INTEGER NOT NULL

, addressLine1 VARCHAR2(50) NOT NULL

, addressLine2 VARCHAR2(50) NULL

, townOrCity VARCHAR2(20) NOT NULL

, county VARCHAR2(20) NULL

, country VARCHAR2(20) DEFAULT ('United Kingdom') NOT NULL

, postCode CHAR(10) NULL

, CONSTRAINT pk\_\_custordaddress PRIMARY KEY (id)

);

-- Data

-- first row

INSERT INTO CustOrderAddress VALUES (1, '41 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

DECLARE max\_value NUMBER := 10;

randchr CHAR(1);

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

--insert the values in to the table created above

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '42 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '43 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '44 Park Hill', 'Moseley', 'Birmingham', NULL, 'United Kingdom', 'B13 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '60 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '61 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '62 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '63 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '64 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), to\_char(i) || '65 Primrose Hill', 'Highgate', 'Birmingham', NULL, 'United Kingdom', 'B3 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), 'Unit 1' || to\_char(i), 'Mumford busines park', 'Coseley', NULL, 'United Kingdom', 'B88 8DR');

INSERT INTO CustOrderAddress VALUES ((SELECT MAX(id) + 1 FROM CustOrderAddress), 'Unit 2' || to\_char(i), 'Mumford busines park', 'Coseley', NULL, 'United Kingdom', 'B88 8DR');

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE CustomerOrder (

id INTEGER NOT NULL

, createdDate DATE DEFAULT SYSDATE NOT NULL

, shipDate DATE NULL

, orderTotal DECIMAL(8,2) NOT NULL

, orderStatusId INTEGER NOT NULL

, billingAddressId INTEGER NOT NULL

, shippingAddressId INTEGER NOT NULL

, customerId INTEGER NOT NULL

, CONSTRAINT pk\_\_custorder PRIMARY KEY (id)

, CONSTRAINT fk\_\_custorder\_status FOREIGN KEY (orderStatusId) REFERENCES OrderStatus (id)

, CONSTRAINT fk\_\_custorder\_billaddr FOREIGN KEY (billingAddressId) REFERENCES CustOrderAddress (id)

, CONSTRAINT fk\_\_custorder\_shipaddr FOREIGN KEY (shippingAddressId) REFERENCES CustOrderAddress (id)

/\* Avoid orders with zero or negative values \*/

, CONSTRAINT ck\_\_custorder\_total CHECK (orderTotal > 0.0)

/\* Can't ship earlier than the order was created! \*/

, CONSTRAINT ck\_\_custorder\_shipdate CHECK (shipDate > createdDate)

) /\* added to ClusCustOrder as described above \*/

CLUSTER ClusCustOrder (id);

/\* no index on order status because that is a tiny table of just a few category names

which is faster to search linearly \*/

-- Data

-- first row

INSERT INTO CustomerOrder VALUES (1, '5-Mar-2016', '20-Mar-2016', 1000.0, 2, 1, 10, 1);

DECLARE max\_value NUMBER := 100;

randval INTEGER;

randdec DECIMAL(8,2);

randnor NUMBER;

randbin INTEGER;

randbill INTEGER;

randship INTEGER;

randcust INTEGER;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

SELECT DBMS\_RANDOM.NORMAL

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(billingAddressId) FROM Customer)))

INTO randnor, randbill FROM Dual;

SELECT trunc(DBMS\_RANDOM.VALUE(-1000, 1000))

, round(DBMS\_RANDOM.VALUE(200, 5000), 2)

, CASE WHEN randnor > 0 THEN 2 ELSE 1 END

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(id) + 1 FROM Customer)))

-- unusual case is a different shipping id

, CASE WHEN randnor > 1 THEN trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(billingAddressId) FROM Customer))) ELSE randbill END

INTO randval, randdec, randbin, randcust, randship FROM Dual;

INSERT INTO CustomerOrder VALUES ((SELECT MAX(id) + 1 FROM CustomerOrder)

, (SELECT to\_date('2013-01-01', 'yyyy-mm-dd') + randval FROM Dual)

, (SELECT to\_date('2013-01-14', 'yyyy-mm-dd') + randval FROM Dual)

, randdec, randbin, randbill, randship, randcust);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE WorksOrder (

id INTEGER NOT NULL

, createdDate DATE DEFAULT SYSDATE NOT NULL

, requiredDate DATE NOT NULL

, assignedToId INTEGER NOT NULL

, completedDate DATE NULL

, completedById INTEGER NULL

, customerOrderId INTEGER NOT NULL

, CONSTRAINT pk\_\_workorder PRIMARY KEY (id)

, CONSTRAINT fk\_\_workorder\_assignEmp FOREIGN KEY (assignedToId) REFERENCES Employee (id)

, CONSTRAINT fk\_\_workorder\_completEmp FOREIGN KEY (completedById) REFERENCES Employee (id)

, CONSTRAINT fk\_\_workorder\_custorder FOREIGN KEY (customerOrderId) REFERENCES CustomerOrder (id)

, CONSTRAINT ck\_\_workorder\_reqdate CHECK (requiredDate > createdDate)

, CONSTRAINT ck\_\_workorder\_compdate CHECK (completedDate > createdDate)

/\* Adding a unique index on custOrderId enforces

the one to one relationship with CustomerOrder

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

, CONSTRAINT uq\_\_workorder\_custorder UNIQUE (customerOrderId)

) /\* added to ClusCustOrder as described above \*/

CLUSTER ClusCustOrder (customerOrderId);

/\* Index on FK to CustOrder is desirable as this table will often be joined to CO for queries \*/

/\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

/\* indices on FK to emp table (there are two such columns)

because it is realistic to have queries over

which employees were assigned and completed works orders \*/

CREATE INDEX idx\_\_workorder\_assignEmp ON WorksOrder (assignedToId);

CREATE INDEX idx\_\_workorder\_completEmp ON WorksOrder (completedById);

-- Data

-- first row

INSERT INTO WorksOrder VALUES (1, '6-Mar-2016', '25-Mar-2016', 1, '18-Mar-2016', 1, 1);

DECLARE max\_value NUMBER := 80;

randnor NUMBER;

randemp1 INTEGER;

randemp2 INTEGER;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

SELECT DBMS\_RANDOM.NORMAL

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(id) FROM Employee)))

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(id) FROM Employee)))

INTO randnor, randemp1, randemp2 FROM Dual;

INSERT INTO WorksOrder VALUES (i + 1

, (SELECT createdDate + 3 FROM CustomerOrder WHERE id = i + 1)

, (SELECT shipDate + 4 FROM CustomerOrder WHERE id = i + 1)

, randemp1

, (SELECT shipDate - 2 FROM CustomerOrder WHERE id = i + 1)

, CASE WHEN randnor > 1 THEN randemp2 ELSE randemp1 END

, i + 1);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Invoice (

id INTEGER NOT NULL

, createdDate DATE DEFAULT SYSDATE NOT NULL

, dueDate DATE DEFAULT SYSDATE + 30 NOT NULL

, worksOrderId INTEGER NOT NULL

, CONSTRAINT pk\_\_invoice PRIMARY KEY (id)

, CONSTRAINT fk\_\_invoice\_workorder FOREIGN KEY (worksOrderId) REFERENCES WorksOrder (id)

/\* Adding a unique index on worksOrderId enforces

the one to one relationship with WorksOrder

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

, CONSTRAINT uq\_\_invoice\_workorder UNIQUE (worksOrderId)

) /\* Archive old Invoices every 5 years because they're no longer accessed after an audit \*/

PARTITION BY RANGE (createdDate)

(PARTITION Inv\_Archive2010 VALUES LESS THAN ('1-Jan-2010')

, PARTITION Inv\_Archive2015 VALUES LESS THAN ('1-Jan-2015')

, PARTITION Inv\_Archive2020 VALUES LESS THAN ('1-Jan-2020'))

;

/\* Index on FK to WorksOrder is desirable as this table will often be joined to WO for queries \*/

/\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

-- Data

-- first row

INSERT INTO Invoice VALUES (1, '18-Mar-2016', '18-Apr-2016', 1);

DECLARE max\_value NUMBER := 80;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

INSERT INTO Invoice VALUES (i + 1

, (SELECT createdDate + 15 FROM WorksOrder WHERE id = i + 1)

, (SELECT createdDate + 45 FROM WorksOrder WHERE id = i + 1)

, i + 1);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Product (

id INTEGER NOT NULL

, name VARCHAR2(100) NOT NULL

, price DECIMAL(8,2) NOT NULL

, instructions VARCHAR2(250)

, CONSTRAINT pk\_\_product PRIMARY KEY (id)

/\* Mustn't have two names the same or it would get messy \*/

, CONSTRAINT uq\_\_product\_name UNIQUE (name)

, CONSTRAINT ck\_\_product\_price CHECK (price >= 0.0)

) /\* added to ClusProduct as described above \*/

CLUSTER ClusProduct (Id);

INSERT INTO Product VALUES (1, 'Oxford Armchair (Green Velvet)', 850.00, 'Take the Oxford frame, 5 foam padding packs and the green velvet upholstery and join it all together with one can of wood glue');

INSERT INTO Product VALUES ((SELECT MAX(id) + 1 FROM Product), 'Knots Landing 3 Piece Suite (Fuchsia Leather)', 1850.00, 'Take the Knots Landing frame pieces (all three of them), the 12 foam padding packs and the fuchsia leather upholstery and join it all together with 3 cans of wood glue');

INSERT INTO Product VALUES ((SELECT MAX(id) + 1 FROM Product), 'Imbiss Kithen Worktop (Brushed Stainless Steel)', 2050.00, 'Take the Imbiss frame pieces (4 in total) and join it all together with 2 cans of wood glue. Apply remaining wood glue to the underside of the worktop and assemble');

INSERT INTO Product VALUES ((SELECT MAX(id) + 1 FROM Product), 'Kiez Sofa Set (Vintage Orange Velvet)', 1809.00, 'Take the Kiez frame pieces (all four of them), the 12 foam padding packs and the orange velvet upholstery and join it all together with 3 cans of wood glue');

COMMIT;

CREATE TABLE CustomerOrderLine (

customerOrderId INTEGER NOT NULL

, productId INTEGER NOT NULL

, quantity INTEGER NOT NULL

/\* Composite Key as this is an n-to-m relationship table \*/

, CONSTRAINT pk\_\_custorderline PRIMARY KEY (customerOrderId, productId)

, CONSTRAINT fk\_\_custorderline\_custorder FOREIGN KEY (customerOrderId) REFERENCES CustomerOrder (id)

, CONSTRAINT fk\_\_custorderline\_prod FOREIGN KEY (productId) REFERENCES Product (id)

, CONSTRAINT ck\_\_custorderline\_qty CHECK (quantity > 0)

) /\* added to ClusCustOrder as described above \*/

CLUSTER ClusCustOrder (customerOrderId);

/\* The two columns used in most searches are already part of the composite key

Won't add any index\*/

-- Data

-- first row

INSERT INTO CustomerOrderLine VALUES (1, 1, 2);

DECLARE max\_value NUMBER := 100;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

INSERT INTO CustomerOrderLine VALUES (i + 1

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(id) + 1 FROM Product)))

, trunc(DBMS\_RANDOM.VALUE(1, 4)));

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE StockItem (

id INTEGER NOT NULL

, name VARCHAR2(100) NOT NULL

, goodsInQty INTEGER DEFAULT (0) NOT NULL

, warehouseQty INTEGER DEFAULT (0) NOT NULL

, workshopQty INTEGER DEFAULT (0) NOT NULL

, CONSTRAINT pk\_\_stktem PRIMARY KEY (id)

/\* Mustn't have two names the same or it would get messy \*/

, CONSTRAINT uq\_\_stktem\_name UNIQUE (name)

/\* can never be fewer than zero \*/

, CONSTRAINT ck\_\_stktem\_ginqty CHECK (goodsInQty >= 0)

, CONSTRAINT ck\_\_stktem\_whsqty CHECK (warehouseQty >= 0)

, CONSTRAINT ck\_\_stktem\_wksqty CHECK (workshopQty >= 0)

);

INSERT INTO StockItem VALUES (1, 'Wood Glue', 0, 2, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Foam Padding Pack', 0, 20, 10);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Oxford Chair Frame', 0, 0, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Oxford Chair Green Velvet Upholstery', 0, 2, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Knots Landing 3 Piece Suite Frame', 0, 1, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Knots Landing 3 Piece Suite Fuchsia Leather Upholstery', 0, 1, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Imbiss Frame', 0, 2, 2);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Imbiss Worktop Piece', 0, 2, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Kiez Frame', 1, 1, 1);

INSERT INTO StockItem VALUES ((SELECT MAX(id) + 1 FROM StockItem), 'Kiez Orange Velvet Upholstery', 1, 1, 1);

COMMIT;

CREATE TABLE ProductStockItem (

productId INTEGER NOT NULL

, stockItemId INTEGER NOT NULL

, quantity INTEGER NOT NULL

/\* Composite Key as this is an n-to-m relationship table \*/

, CONSTRAINT pk\_\_prodstktem PRIMARY KEY (productId, stockItemId)

, CONSTRAINT fk\_\_prodstktem\_prod FOREIGN KEY (productId) REFERENCES Product (id)

, CONSTRAINT fk\_\_prodstktem\_stktem FOREIGN KEY (stockItemId) REFERENCES StockItem (id)

, CONSTRAINT ck\_\_prodstktem\_qty CHECK (quantity > 0)

) /\* added to ClusProduct as described above \*/

CLUSTER ClusProduct (ProductId);

/\* The two columns used in most searches are already part of the composite key

Won't add any index\*/

INSERT INTO ProductStockItem VALUES (1, 1, 1);

INSERT INTO ProductStockItem VALUES (1, 2, 5);

INSERT INTO ProductStockItem VALUES (1, 3, 1);

INSERT INTO ProductStockItem VALUES (1, 4, 1);

INSERT INTO ProductStockItem VALUES (2, 1, 3);

INSERT INTO ProductStockItem VALUES (2, 2, 12);

INSERT INTO ProductStockItem VALUES (2, 5, 1);

INSERT INTO ProductStockItem VALUES (2, 6, 1);

INSERT INTO ProductStockItem VALUES (3, 7, 2);

INSERT INTO ProductStockItem VALUES (3, 8, 1);

INSERT INTO ProductStockItem VALUES (3, 1, 3);

INSERT INTO ProductStockItem VALUES (4, 9, 4);

INSERT INTO ProductStockItem VALUES (4, 10, 1);

INSERT INTO ProductStockItem VALUES (4, 1, 3);

COMMIT;

CREATE TABLE Supplier (

id INTEGER NOT NULL

, companyName VARCHAR2(50)

, companyAddrId INTEGER NOT NULL

, mainContactId INTEGER NOT NULL

, CONSTRAINT pk\_\_supplier PRIMARY KEY (id)

/\* Mustn't have two names the same or it would get messy \*/

, CONSTRAINT uq\_\_supplier\_name UNIQUE (companyName)

, CONSTRAINT fk\_\_supplier\_address FOREIGN KEY (companyAddrId) REFERENCES Address (id)

, CONSTRAINT fk\_\_supplier\_contact FOREIGN KEY (mainContactId) REFERENCES Person (id)

/\* No unique on supplier main contact. It's feasible that the same person runs more than one company in the same industry \*/

);

/\* index on FK to Person table because looking up the contact details will be a common query \*/

CREATE INDEX idx\_\_supplier\_contact ON Supplier (mainContactId);

INSERT INTO Supplier VALUES (1, 'Oak and Fold Partners', (SELECT MAX(id) FROM Address), (SELECT MAX(id) FROM Person));

INSERT INTO Supplier VALUES ((SELECT MAX(id) + 1 FROM Supplier)

, 'Wooken Legs Ltd', (SELECT MAX(id) - 1 FROM Address), (SELECT MAX(id) - 1 FROM Person));

INSERT INTO Supplier VALUES ((SELECT MAX(id) + 1 FROM Supplier)

, 'Hokem Pokem and Sons - Bits n Bobs', (SELECT MAX(id) - 2 FROM Address), (SELECT MAX(id) - 2 FROM Person));

INSERT INTO Supplier VALUES ((SELECT MAX(id) + 1 FROM Supplier)

, 'Lots Wife of Stuff', (SELECT MAX(id) - 3 FROM Address), (SELECT MAX(id) - 3 FROM Person));

INSERT INTO Supplier VALUES ((SELECT MAX(id) + 1 FROM Supplier)

, 'Hairy Marys Emprorium', (SELECT MAX(id) - 4 FROM Address), (SELECT MAX(id) - 4 FROM Person));

CREATE TABLE StockItemSupplier (

stockItemId INTEGER NOT NULL

, supplierId INTEGER NOT NULL

, agreedPrice DECIMAL(8,2) NOT NULL

/\* Composite Key as this is an n-to-m relationship table \*/

, CONSTRAINT pk\_\_stktemsupp PRIMARY KEY (stockItemId, supplierId)

, CONSTRAINT fk\_\_stktemsupp\_stktem FOREIGN KEY (stockItemId) REFERENCES StockItem (id)

, CONSTRAINT fk\_\_stktemsupp\_supp FOREIGN KEY (supplierId) REFERENCES Supplier (id)

, CONSTRAINT ck\_\_stktemsupp\_agprice CHECK (agreedPrice >= 0.0)

);

/\* The two columns used in most searches are already part of the composite key

Won't add any index\*/

INSERT INTO StockItemSupplier VALUES (1, 1, 20.00);

INSERT INTO StockItemSupplier VALUES (1, 2, 25.00);

INSERT INTO StockItemSupplier VALUES (2, 1, 300.00);

INSERT INTO StockItemSupplier VALUES (2, 2, 305.00);

INSERT INTO StockItemSupplier VALUES (1, 3, 150.00);

INSERT INTO StockItemSupplier VALUES (1, 4, 140.00);

INSERT INTO StockItemSupplier VALUES (3, 2, 500.00);

INSERT INTO StockItemSupplier VALUES (4, 2, 510.00);

INSERT INTO StockItemSupplier VALUES (3, 3, 499.00);

INSERT INTO StockItemSupplier VALUES (4, 3, 500.00);

INSERT INTO StockItemSupplier VALUES (3, 4, 500.00);

INSERT INTO StockItemSupplier VALUES (4, 4, 510.00);

INSERT INTO StockItemSupplier VALUES (5, 3, 499.00);

INSERT INTO StockItemSupplier VALUES (6, 3, 500.00);

INSERT INTO StockItemSupplier VALUES (5, 4, 500.00);

INSERT INTO StockItemSupplier VALUES (6, 4, 510.00);

INSERT INTO StockItemSupplier VALUES (7, 3, 200.00);

INSERT INTO StockItemSupplier VALUES (7, 4, 205.00);

INSERT INTO StockItemSupplier VALUES (8, 3, 499.00);

INSERT INTO StockItemSupplier VALUES (8, 4, 500.00);

INSERT INTO StockItemSupplier VALUES (9, 5, 500.00);

INSERT INTO StockItemSupplier VALUES (10, 5, 205.00);

COMMIT;

CREATE TABLE PurchaseOrder (

id INTEGER NOT NULL

, createdDate DATE DEFAULT SYSDATE NOT NULL

, supplierId INTEGER NOT NULL

, orderTotal NUMBER(5) NOT NULL

, orderStatusId INTEGER NOT NULL

, completedDate DATE DEFAULT SYSDATE NULL

, CONSTRAINT pk\_\_purchorder PRIMARY KEY (id)

, CONSTRAINT fk\_\_purchord\_supp FOREIGN KEY (supplierId) REFERENCES Supplier (id)

, CONSTRAINT fk\_\_purchorder\_status FOREIGN KEY (orderStatusId) REFERENCES OrderStatus (id)

/\* Avoid orders with zero or negative values \*/

, CONSTRAINT ck\_\_purchorder\_total CHECK (orderTotal >= 0.0)

/\* Can't complete earlier than the order was created! \*/

, CONSTRAINT ck\_\_purchorder\_completdate CHECK (completedDate > createdDate)

) /\* Archive old POs every 5 years because they're no longer accessed after an audit \*/

PARTITION BY RANGE (createdDate)

(PARTITION PO\_Archive2010 VALUES LESS THAN ('1-Jan-2010')

, PARTITION PO\_Archive2015 VALUES LESS THAN ('1-Jan-2015')

, PARTITION PO\_Archive2020 VALUES LESS THAN ('1-Jan-2020'))

;

/\* no index on order status because that is a tiny table of just a few category names

which is faster to search linearly \*/

/\* index on FK to Supplier table because join to supplier table will be very common \*/

CREATE INDEX idx\_\_purchord\_supp ON PurchaseOrder (supplierId);

-- Data

-- First row

INSERT INTO PurchaseOrder VALUES (1, '1-May-2016', 1, 300.00, 2, '20-May-2016');

DECLARE max\_value NUMBER := 80;

randval NUMBER;

randnor NUMBER;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

SELECT DBMS\_RANDOM.NORMAL

, trunc(DBMS\_RANDOM.VALUE(-1000, 1000))

INTO randnor, randval FROM Dual;

INSERT INTO PurchaseOrder VALUES (i + 1

, (SELECT to\_date('2013-01-01', 'yyyy-mm-dd') + randval FROM Dual)

, trunc(DBMS\_RANDOM.VALUE(1, (SELECT MAX(id) + 1 FROM Supplier)))

, 1000.00

, CASE WHEN randnor > 0 THEN 2 ELSE 1 END

, (SELECT to\_date('2013-01-20', 'yyyy-mm-dd') + randval FROM Dual)

);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE PurchaseOrderLine (

purchOrderId INTEGER NOT NULL

, stockItemId INTEGER NOT NULL

, quantity INTEGER NOT NULL

, purchasePrice DECIMAL(8,2) NOT NULL

/\* Composite Key as this is an n-to-m relationship table \*/

, CONSTRAINT pk\_\_purchordline PRIMARY KEY (purchOrderId, stockItemId)

, CONSTRAINT fk\_\_purchordline\_purchord FOREIGN KEY (purchOrderId) REFERENCES PurchaseOrder (id)

, CONSTRAINT fk\_\_purchordline\_stktem FOREIGN KEY (stockItemId) REFERENCES StockItem (id)

, CONSTRAINT ck\_\_purchordline\_qty CHECK (quantity > 0)

, CONSTRAINT ck\_\_purchordline\_purchprice CHECK (purchasePrice >= 0.0)

);

/\* The two columns used in most searches are already part of the composite key

Won't add any index\*/

-- Data

-- First Row

INSERT INTO PurchaseOrderLine VALUES (1, 1, 2, 2 \* (SELECT MIN(agreedPrice) FROM StockItemSupplier WHERE stockItemId = 1));

DECLARE max\_value NUMBER := 80;

randite1 NUMBER;

randite2 NUMBER;

randqty1 NUMBER;

randqty2 NUMBER;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

SELECT trunc(DBMS\_RANDOM.VALUE(1, 5))

, trunc(DBMS\_RANDOM.VALUE(6, 10))

, trunc(DBMS\_RANDOM.VALUE(1, 5))

, trunc(DBMS\_RANDOM.VALUE(1, 5))

INTO randite1, randite2, randqty1, randqty2 FROM Dual;

INSERT INTO PurchaseOrderLine VALUES (i + 1, randite1, randqty1, randqty1 \* (SELECT MIN(agreedPrice) FROM StockItemSupplier WHERE stockItemId = randite1));

INSERT INTO PurchaseOrderLine VALUES (i + 1, randite2, randqty2, randqty2 \* (SELECT MIN(agreedPrice) FROM StockItemSupplier WHERE stockItemId = randite2));

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

COMMIT;

CREATE TABLE CashTransaction (

id INTEGER NOT NULL

, amount DECIMAL(8,2) NOT NULL

, createDate DATE DEFAULT SYSDATE NOT NULL

, processedDate DATE NULL

, bankDate DATE NULL

, CONSTRAINT pk\_\_cashtrans PRIMARY KEY (id)

) /\* added to ClusCashTrans as described above \*/

CLUSTER ClusCashTrans (Id);

-- Data

-- First row

INSERT INTO CashTransaction VALUES (1, 600.00, '30-May-2016', '31-May-2016', '4-Jun-2016');

DECLARE max\_value NUMBER := 100;

randval NUMBER;

randnor NUMBER;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

SELECT trunc(DBMS\_RANDOM.VALUE(-1000, 1000))

INTO randval FROM Dual;

INSERT INTO CashTransaction VALUES (i + 1

, 500.00

, (SELECT to\_date('2013-01-01', 'yyyy-mm-dd') + randval FROM Dual)

, (SELECT to\_date('2013-01-03', 'yyyy-mm-dd') + randval FROM Dual)

, (SELECT to\_date('2013-01-08', 'yyyy-mm-dd') + randval FROM Dual)

);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Receipt (

cashTransId INTEGER NOT NULL

, invoiceId INTEGER NOT NULL

, CONSTRAINT pk\_\_receipt PRIMARY KEY (cashTransId, invoiceId)

/\* Adding a unique index on cashTransId enforces

the one to one relationship with CashTransaction

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

, CONSTRAINT uq\_\_receipt\_cashtrans UNIQUE (cashTransId)

, CONSTRAINT fk\_\_receipt\_cashtrans FOREIGN KEY (cashTransId) REFERENCES CashTransaction (id)

, CONSTRAINT fk\_\_receipt\_inv FOREIGN KEY (invoiceId) REFERENCES Invoice (id)

) /\* added to ClusCashTrans as described above \*/

CLUSTER ClusCashTrans (cashTransId);

/\* Index on FK to CashTrans is desirable as this table will often be joined to CT for queries \*/

/\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

/\* indices on FK to invoice because it is realistic to have queries

which look up specific invoices from cash receipts \*/

CREATE INDEX idx\_\_receipt\_inv ON Receipt (invoiceId);

-- Data

DECLARE max\_value NUMBER := 81;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

INSERT INTO Receipt VALUES (i, i);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

CREATE TABLE Payment (

cashTransId INTEGER NOT NULL

, purchOrderId INTEGER NOT NULL

, CONSTRAINT pk\_\_payment PRIMARY KEY (cashTransId, purchOrderId)

/\* Adding a unique index on cashTransId enforces

the one to one relationship with CashTransaction

and also speeds up queries joining the two tables

which is essential as it will happen all the time \*/

, CONSTRAINT uq\_\_payment\_cashtrans UNIQUE (cashTransId)

, CONSTRAINT fk\_\_payment\_cashtrans FOREIGN KEY (cashTransId) REFERENCES CashTransaction (id)

, CONSTRAINT fk\_\_payment\_purchord FOREIGN KEY (purchOrderId) REFERENCES PurchaseOrder (id)

) /\* added to ClusCashTrans as described above \*/

CLUSTER ClusCashTrans (cashTransId);

/\* Index on FK to CashTrans is desirable as this table will often be joined to CT for queries \*/

/\* UQ Constraint already created an index - tested by trying to create an explicit index. \*/

/\* indices on FK to invoice because it is realistic to have queries

which look up specific purchase orders from payments \*/

CREATE INDEX idx\_\_payment\_purch ON Payment (purchOrderId);

-- Data

DECLARE max\_value NUMBER := 20;

--start of script

BEGIN

--go round the loop max\_value times - value supplied by the user

FOR i IN 1..max\_value LOOP

INSERT INTO Payment VALUES (i + 81, i);

END LOOP;

-- end of script

END;

--need to enter a new line (ENTER) to make script run

/

COMMIT;

/\* Users, Roles, Views and Security \*/

CREATE VIEW v\_CustomerDetails AS

SELECT c.id AS "Customer No."

, p.givenName AS "Given Name"

, p.familyName AS "Family Name"

, p.dateOfBirth AS "Date of Birth"

, p.email AS "Email Address"

FROM Person p

INNER JOIN Customer c

ON p.id = c.personId;

CREATE VIEW v\_CustomerOrderDetails AS

SELECT c.id AS "Customer No."

, p.givenName AS "Given Name"

, p.familyName AS "Family Name"

, p.email AS "Email Address"

, co.createdDate AS "Order Date"

, co.shipDate AS "Ship Date"

, co.orderTotal AS "Order Total"

, os.statusName AS "Order Status"

FROM Person p

INNER JOIN Customer c

ON p.id = c.personId

INNER JOIN CustomerOrder co

ON c.id = co.customerId

INNER JOIN OrderStatus os

ON co.orderStatusId = os.id;

CREATE VIEW v\_WorksOrdersListing AS

SELECT wo.id AS "Works Order No."

, co.id AS "Customer Order No."

, co.customerId AS "Customer No."

, wo.createdDate AS "Order Date"

, wo.requiredDate AS "Required Date"

, p.givenName || ' ' || p.familyName AS "Assigned To"

, wo.completedDate AS "Completed Date"

, p2.givenName || ' ' || p2.familyName AS "Completed By"

FROM WorksOrder wo

INNER JOIN CustomerOrder co

ON wo.customerOrderId = co.id

INNER JOIN Employee e

ON wo.assignedToId = e.id

INNER JOIN Person p

ON e.personId = p.id

LEFT OUTER JOIN Employee e2

ON wo.completedById = e2.id

LEFT OUTER JOIN Person p2

ON e2.personId = p2.id;

CREATE VIEW v\_InventoryOverview AS

SELECT si.id AS "SKU No."

, si.name "Item Description"

, SUM(si.goodsInQty + si.warehouseQty + si.workshopQty) AS "Inventory Quantity"

, nvl(foq.ForwardOrdersQuantity, 0) AS "Forward Orders"

FROM StockItem si

LEFT OUTER JOIN (

SELECT psi.stockItemId

, col.quantity \* psi.quantity AS ForwardOrdersQuantity

FROM ProductStockItem psi

INNER JOIN Product p

ON psi.productId = p.id

INNER JOIN CustomerOrderLine col

ON p.id = col.productId

INNER JOIN CustomerOrder co

ON col.customerOrderId = co.id

INNER JOIN OrderStatus os

ON co.orderStatusId = os.id

WHERE os.statusName = 'Pending'

) foq

ON si.id = foq.stockItemId

GROUP BY si.id

, si.name

, si.goodsInQty

, si.warehouseQty

, si.workshopQty

, foq.ForwardOrdersQuantity;