QlikView

Generating missing data

QlikView Technical Brief

4 Feb 2013, HIC

www.qlikview.com

QlikView

Contents

Contents 2

Introduction 3
- Cases from real life 3

Basic table generation 4
- Generate a table using Load … Resident 4
  - *Example: Master Calendar using Load Resident* 4
- Generate a table using Load … Autogenerate 4
  - *Example: Master Calendar using Autogenerate* 5

The Peek function 6
- *Example: Propagate a value downwards* 6
- *Example: Accumulate a number* 6
- Populating a sparsely populated field using Generate, Join and Peek 7
  - *Example: Populating a table with conversion rates* 7

Loops inside the Load statement 9
- While and Iterno() 9
  - *Example: Creating one record per day that a contract is valid* 9
- The Subfield function 10
  - *Example: Creating one record per skill from a list of skills* 11

Generating all combinations of several fields 12
- Cartesian product using Join 12
  - *Example: Populating a table with warehouse balances* 12
- Intervalmatch 14

Simulations in QlikView 15
- *Example: Monte-Carlo simulation of throwing two dice* 15
- *Example: Monte-Carlo simulation of initial poker hand* 16

QlikView

Introduction

Often when you create a QlikView data model, you need to *create* data in the script. It could be that

- an entire table is missing but that it can be inferred from other data.
- some records are missing, but that common sense tells you that they should be there so you want to generate them so the corresponding values become clickable in the QlikView GUI. One situation is that you want to create several records from one single existing record.
- the records exist, but you want to propagate a field value from the record immediately above.

In all these cases, you need to generate data in the QlikView script. This Technical Brief is about different methods to do this.

Cases from real life

Conversion rates

The source data is a table that lists currency conversion rates. However, this table only contains the dates where the conversion rate changed, not the dates between the changes. Then you need to generate the dates between the changes as individual records and use the value from the previous date.

Warehouse balance

Just as in the example above, only the changes are stored in the source data, and you need to generate the records that correspond to the days between the changes and in these new records use the balance of the last change. But in this example, you need to do this for each product in the warehouse, so that you have a balance for each combination of product and date.

Contracts with a limited validity in time

The source data has a table which is a list of contracts; one contract per record. Each record contains a “Begin” date and an “End” date that defines the validity time of the contract. The user will want to ask questions like “How many valid contracts do I have a given day?” To answer this, you need to generate all dates between the beginning and the end of the validity interval – you need to loop over the existing record – so that it is possible to click on the date in order to make this selection.

Master calendar table

The most common case is however the Master Calendar: In almost all QlikView applications there is a date and from this date you can infer year, month, week day, etc. This is best done as a separate table with date as a key linking to the original data. This table needs to be generated in QlikView and in it you can have many columns for the different calendar entities.

Basic table generation

Generate a table using Load … Resident

One common way to generate a new table is to load one or several fields from an already loaded table – typically the transaction table – using a distinct or group by clause.

One drawback with this method is that the already loaded table might not have all values of the field in question. If you want all values, you should generate the table using autogenerate instead.

Example: Master Calendar using Load Resident

The master calendar is a table that often not exists in the source database, but is needed to hold all fields that can be inferred from date: Month, week no, week day, etc. It can be created in many different ways. The simplest way is to use a Load resident with a distinct clause that picks out all distinct values of a date that exists in a transaction table:

```
MasterCalendar:
Load distinct Date,
    Year(Date) as Year,
    Month(Date) as Month,
    Day(Date) as Day
Resident TransactionTable;
```

The basic load statement creates a table with the distinct values of Date. This field is then in turn used to create fields for Year, Month and Day. Of course other fields like YearMonth, FiscalYear, etc. can be created.

Generate a table using Load … Autogenerate

Another common way to generate data in QlikView is to autogenerate records:

```
Load RecNo() as X Autogenerate 100 ;
```

This construction is very similar to the other possibilities of Load. You can feed the Load statement in different ways:

```
Load … From          <File>        ; (from file)
Load … Resident      <Table>       ; (from an already loaded table)
Load … Inline        <InlineTable> ; (from a table written in the script)
Load … Autogenerate  <Number>      ; (from nothing)
```

But with autogenerate there is no source – the records are generated and all field values must be derived from functions like Rand() or RecNo(). The number of records is specified in the number after the Autogenerate keyword.

Example: Master Calendar using Autogenerate

Another way to create the master calendar is to use autogenerate to generate all dates in a range. To do this, you need to first define the range. One way is to look for the smallest and largest dates in the date field and then generate all the dates in between the two:

```
MinMaxDate:
Load Min(Date) as MinDate, Max(Date) as MaxDate resident TransactionTable;
Let vMinDate = Peek('MinDate',-1,'MinMaxDate') - 1;
Let vMaxDate = Peek('MaxDate',-1,'MinMaxDate')    ;

Drop Table MinMaxDate;

MasterCalendar:
Load Date,
    Year(Date) as Year,
    Month(Date) as Month,
    Day(Date) as Day;
Load Date(recno()+$(vMinDate)) as Date Autogenerate vMaxDate - vMinDate;
```

First the MinMaxDate table is created. It has one line only and contains the largest and smallest dates in the data. These values are stored in two separate variables using Let statements and the Peek() function.

The variables are then used in a following Load statement to generate all dates in the range. Note that a preceding load is used to define all calendar fields, except the primary key “Date”.

The variables can be created in other ways, for instance this year and last year only:

```
Let vMinDate = Floor(YearStart(Today(),-1)) - 1;
Let vMaxDate = Floor(YearEnd(Today()))    ;
```

The Peek function

In all cases where you want to propagate values downward in a loaded table, the Peek() function is my preferred solution. This function returns the value of the preceding record. It can be used to fetch any record from any table, but here we are only interested in fetching the record immediately above, which is also the default behavior of Peek().

Example: Propagate a value downwards

You want to replace NULL values with the value from the above record. Then you should use

```
If( IsNull( Field ), Peek( Field ), Field ) as Field
```

In this example, the condition is a simple “IsNull(Field)” but you can of course have other, much more complex logic here.

In some cases, you want to propagate values downward in the loaded table, but just within some sort of group definition. Further, if the data isn’t sorted you need to do this using an “order by” in the load statement. However, “order by” will only work together with a resident load, i.e. you may need to do the operation in two steps.

Example: Accumulate a number

You have a number and you want to accumulate this amount over time, but only within the same product. In the following example, an additional field with the accumulated amount is created:

```
TempTable_Data:
Load Product, Date, Amount From DataTable ;

Data:
Load Product, Date, Amount,
   If(  Product=Peek(Product),       // if the Product is the same as in the previous row
      RangeSum(Amount,Peek(AccumulatedAmount)),
      RangeSum(Amount)) as AccumulatedAmount
   Resident TempTable_Data
   Order By Product, Date ;          // order by Product and by Date within the Product

Drop Table TempTable_Data ;
```

In this example, two passes are made over the data. The reason is that you need the “order by” and this is only possible within a resident load, i.e. in the second pass.

The Peek() function is used first to check that the record pertains to the same product as the previous record, then a second time to fetch the “AccumulatedAmount” value from the previous record.

Product	Date	Amount	AccumulatedAmount
A	2013-01-20	20	20
A	2013-01-21	NULL	20
A	2013-01-22	40	60
B	2013-01-20	NULL	0
B	2013-01-21	10	10
B	2013-01-22	NULL	10

The RangeSum() function is used to add the two numbers. The reason you need to use this function is that normal addition does not work for NULL values, whereas RangeSum() considers NULL as a zero.

Finally, the temporary data table is dropped.

Populating a sparsely populated field using Generate, Join and Peek

Another common case where you need the Peek() function is when you want to populate a sparsely populated field. Common cases are conversion rates and warehouse balances where only the dates with changing numbers can be found in the data.

In both these cases, you often want to ask the question: "What was the status on this specific day?" In other words, you want to click on a reference date to see the number that is associated with this date – but the date might not exist in the data.

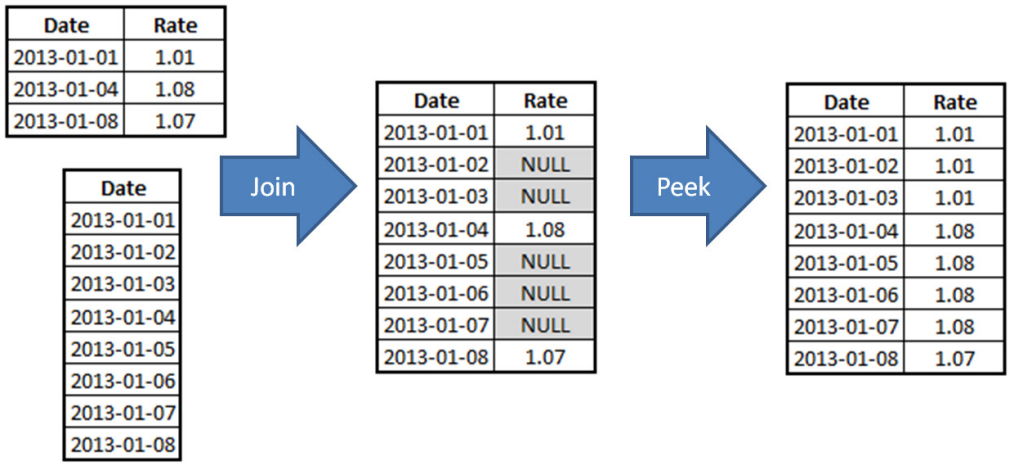
Example: Populating a table with conversion rates

The source data is a table that lists conversion rates. However, the table only contains the dates where the conversion rate changed, not the dates between the changes. Hence, you want to insert missing dates and fill the "Rate" field with the appropriate value. Then you should first generate the days, join them onto the Rate table, sort it according to Date and finally propagate above values downwards, when appropriate:

```
TempTable_Rates:
Load Date, Rate From Rates ;

MinMaxDate:
Load Min(Date) as MinDate, Max(Date) as MaxDate resident TempTable_Rates;
Let vMinDate = Peek('MinDate',-1,'MinMaxDate') - 1;
Let vMaxDate = Peek('MaxDate',-1,'MinMaxDate')    ;

Drop Table MinMaxDate;

Join (TempTable_Rates)
Load Date(recno()+$(vMinDate)) as Date Autogenerate vMaxDate - vMinDate;
```

```
Rates:
NoConcatenate Load Date,
    If(  IsNull( Rate ), Peek( Rate ), Rate ) as Rate
    Resident TempTable_Rates
    Order By Date ; // so that above values can be propagated downwards

Drop Table TempTable_Rates;
```

The picture below illustrates the process.



Loops inside the Load statement

Sometimes when loading data you want to load the same record (with some small variation) several times. It could be that the source data contains a range – an upper bound and a lower bound – and you want a second table that has one record per discrete value in the range. It could also be that one record in the source data contains a list of possible discrete values and you want a second table with record per value in the list. You can do such loops either with a While loop or using the Subfield() function.

While and Iterno()

A loop inside the Load statement can be created using the While clause:

```
Load Date, IterNo() as Iteration From … While IterNo() <= 4 ;
```

Such a Load statement will loop over each input record and load this over and over as long as the expression in the While clause is true. The IterNo() function returns “1” in the first iteration, “2” in the second, etc.

The While clause can be combined with any of the source possibilities:

```
Load … From          <File>          While <Expression> ;
Load … Resident      <Table>         While <Expression> ;
Load … Inline        <InlineTable>   While <Expression> ;
Load … Autogenerate  <Number>        While <Expression> ;
Load … While         <Expression> ;  Load …
```

A Load statement with a While clause cannot at the same time have a Where clause. The reason is that it would be unclear which of the two clauses should be evaluated first. If you want to combine them, you should use a preceding Load. I.e., if you want to loop over only the records that fulfill the Where condition, you should use the following construction:

```
Load … While <Expression> ; Load … From <File> Where <Expression> ;
```

And if you want to loop over all records, but just keep the ones that fulfill the where condition, you should use the following construction:

```
Load … Where <Expression> ; Load … From <File> While <Expression> ;
```

The clause in the second of the two Loads will be evaluated first and the result will be piped into the first Load. Which one to choose depends on which precedence you want: Should the filter of the Where clause be applied before or after the loop?

Example: Creating one record per day that a contract is valid

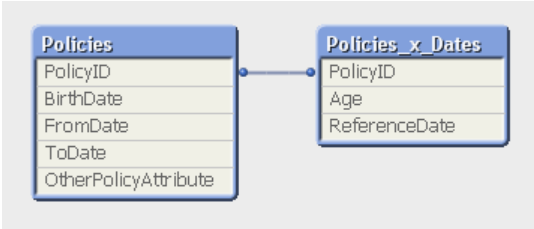
In this example, the source data is a table that lists a number of contracts. Each contract has a begin day and an end day. It could for instance be insurance policies, where an insurance

policy is valid a limited time. The analysts of the insurance company would then probably want to ask the question: “How many valid contracts did we have on this specific day?” In other words, you want to click on a reference date to see the count of insurance policies that are associated with this date – but this date might not exist in the source data.

Then you need to first load all policies in one table and link this to a table that contains one record per contract and date. The second table is generated using a While loop that loads not only the “From” date and the “To” date, but also all dates in between:

```
Policies:
Load PolicyID, BirthDate, FromDate, ToDate, OtherPolicyAttribute
    From Policies;

Policies_x_Dates:
Load PolicyID,
    Age( FromDate + IterNo() – 1, BirthDate ) as Age,
    Date( FromDate + IterNo() – 1 ) as ReferenceDate
    Resident Policies
    While IterNo() <= ToDate - FromDate + 1 ;
```

Note that the Policies table has exactly one record per insurance policy, and the newly created Policies_x_Dates table has exactly one record per combination of policy and date. Note also that there are other fields that should be put in the Policies_x_Dates table, e.g., the age of the insured person, since it depends on the reference date.



The Subfield function

If you have list of values in one field and you want to split the record into several records (have one record per value in the list) you should instead use the subfield function.

```
Load RecordID, Subfield( ListOfValues, '|') as Value Resident … ;
```

Such a Load statement will loop over each input record and load it several times, once for each value in the list of values. The return value of the Subfield() function will be the n:th value in the list.

The second parameter of the Subfield() function defines the separator of the list. It is possible to have a third parameter in the Subfield() function, but then the function will lose its looping functionality.

Example: Creating one record per skill from a list of skills

The source data is a table that lists a number of skills per employee.

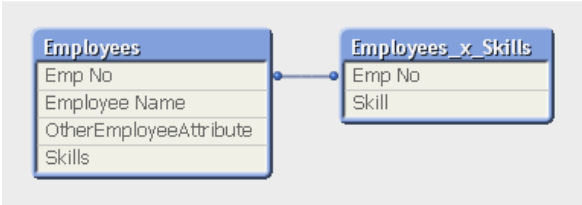
Emp No	Skills
159	Economics,Pharmacology
163	Marketing,Sociology
174	Bookkeeping,Particle physics
210	Economics,Marketing
215	Economics,Law,Marketing
279	Law,Marketing
286	Finance,Marketing
300	Bookkeeping,Finance

The number of individual skills as well as the order is arbitrary. The goal is to have the individual skills in a separate field.

Then you should first load all employees in one table with the list of skills in one field. This table should be linked to a second table that has a field with the individual skills. The second table is generated using a Subfield() call that makes the Load statement loop over the list of skills:

```
Employees:
Load [Emp No], [Employee Name], OtherEmployeeAttribute, Skills
    From Employees;

Employees_x_Skills:
Load [Emp No],
    Trim(Subfield( Skills, ',' )) as Skill
    Resident Employees;
```

The Trim() function removes unwanted extra leading or trailing blanks that may exist inside the list of skills.



Generating all combinations of several fields

Cartesian product using Join

Sometimes you need to compare two or more fields and generate all possible combinations between them. In SQL, you can easily do this using a Cartesian product:

```
SQL SELECT Table1.A, Table2.B FROM Table1, Table2;
```

It is basically a join, but without joining condition.

This will create all combinations of the two fields, without any limitation from other relationships, and you will get a table that most likely has many more records than any of the two individual tables.

Using Load statements, you can do the same:

```
Load A From Table1.csv ;

Join
Load B From Table2.csv ;
```

Once you have made the join, you have a new table that you can process further, e.g. only select some records that fulfill specific demands or generate new fields based on the initial ones. However, further processing must be done in a second pass using a resident Load. Finally the initial table must be dropped.

Example: Populating a table with warehouse balances

In this example, the source data is a table that lists the balances of a number of products in a warehouse over a number of dates. However, only records where the balance has changed exist. This means that for a specific product, there may be dates missing. For these dates, the latest balance should be used.

The example is similar to the previous example on conversion rates, but with the difference that there exist several products and each product has its own series of dates and balances. Hence we now have a two-dimensional problem: Each combination of product and date should exist in the table.

In such a case you should first load all existing product balances (step “A” below). The second step is to generate all combinations of product and date using a join (“B” below).

The third step is to run through all these combinations picking out the missing ones (using “Where Not Exists()”) and appending these to the initial product balance table (“C” below). Then you need make an additional pass in the ordered product balance table, so that you can propagate the appropriate values downwards (“D” below).

And finally, you need to drop the temporary tables (“E” below).

```
// ----   A: Load all existing product balances
TempProductBalances:
Load ProductID, Date, Balance,
    ProductID & '|' & Num( Date ) as Product_x_DateID
    From ProductBalances;

// ----   B: Create all combinations of product and date
TempProduct_x_Dates:
Load distinct ProductID Resident TempProductBalances;

Join (TempProduct_x_Dates)
Load Date(recno()+$(vMinDate)) as Date Autogenerate vMaxDate - vMinDate;

// ----   C: Append missing records onto the product balance table
Concatenate (TempProductBalances)
Load * Where not Exists( Product_x_DateID );
Load ProductID, Date,
    ProductID & '|' & Num( Date ) as Product_x_DateID
    Resident TempProduct_x_Dates ;

// ----   D: Create final product balance table. Propagate value from above record.
ProductBalances:
NoConcatenate
Load ProductID, Date,
    If(  ProductID=Peek( ProductID ) and IsNull( Balance ),
        Peek( Balance ),
        RangeSum( Balance )) as Balance
    Resident TempProductBalances
    Order By ProductID, Date; // so that above values can be propagated downwards

// ----   E: Drop all temporary tables
Drop Table TempProduct_x_Dates, TempProductBalances;
```

Intervalmatch

A special case is when you need to generate all combinations between a numeric field, e.g. the date of an event or a transaction and numeric intervals defined in another table. In SQL, you would solve this by joining the two tables and use a BETWEEN condition:

```
SQL SELECT Events.Date, Intervals.BeginDate, Intervals.EndDate
    FROM Events, Intervals
    WHERE Events.Date BETWEEN Intervals.BeginDate AND Intervals.EndDate;
```

In QlikView you would normally use the IntervalMatch prefix to solve this problem. The general structure of the script would be to first load the events table and the intervals table as they are, and then generate a third table defining a bridge between the two.

```
Events:
Load TransactionID, Date, <OtherEventFields> From Events;

Intervals:
Load IntervalName, BeginDate, EndDate, <OtherIntervalFields>  From Intervals;

IntervalMatchBridge:
IntervalMatch (Date)
Load distinct BeginDate, EndDate Resident Intervals;
```

The intervalmatch will compare the intervals defined by BeginDate and EndDate with the discrete values of Date and generate all combinations.

Note that with IntervalMatch you will get a synthetic key in your data model. This is nothing you need to worry about. Intervalmatch is one of the cases where a synthetic key is the most efficient way of modeling the data. In fact, BeginDate and EndDate together form a primary key for the intervals, so it is quite natural to have them form a synthetic key.

Simulations in QlikView

Using all the above techniques, it is fairly straightforward to make simulations in QlikView. You can combine autogenerate and while loops to create data sets on which you make statistical analysis.

When doing so, there are some functions that are very useful:

- RecNo() – the record number of the input record
- RowNo() – the record number of the output record
- Rand() – a generator of random numbers,
- Ceil() – round upwards to nearest integer,
- Pick() – pick a specific value in a list of values

A small note of warning: If you are to use the result of the simulation for anything relevant, you need to be aware of the uncertainties (statistical errors) of the result, which can be calculated using standard statistical methods.

If you want to use an empirical approach to get a feeling for how large the uncertainties are, just run the script several times to see how much a value changes from time to time.

Example: Monte-Carlo simulation of throwing two dice

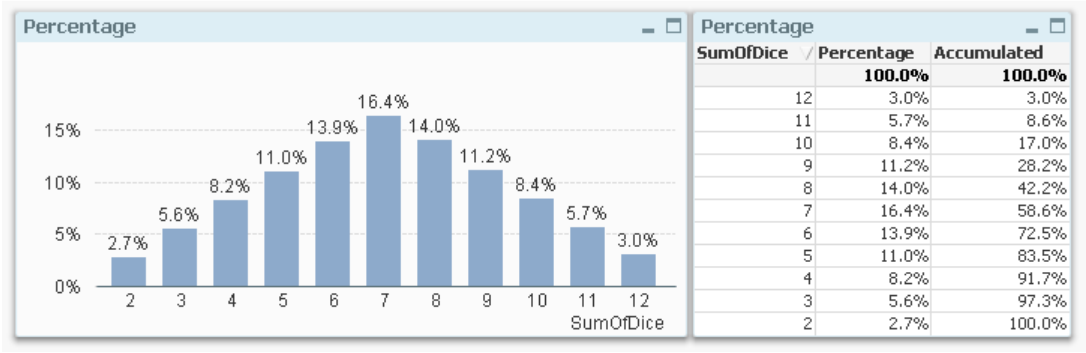
What is the chance of getting a sum higher than a specific number when throwing two dice? To simulate this, you need to generate a large number of throws and randomly create the result of throwing two dice:

```
DiceThrowing:
Load *,
    Dice1 + Dice2 as SumOfDice;
Load
    RecNo( ) as ThrowNo,
    Ceil( Rand( ) * 6 ) as Dice1,
    Ceil( Rand( ) * 6 ) as Dice2
    Autogenerate 100000;
```

This script will autogenerate 100000 throws and store the result of each throw in the field SumOfDice. The result can then be analyzed in a normal QlikView chart.

Below I have a bar chart and a straight table sorted descending showing the result. As formulae, I have used

```
Percentage = Count ( ThrowNo ) / Count ( total ThrowNo )
Accumulated = RangeSum( Above( Accumulated ), Percentage )
```



Percentage

SumOfDice	Percentage	Accumulated
	100.0%	**100.0%**
12	3.0%	3.0%
11	5.7%	8.6%
10	8.4%	17.0%
9	11.2%	28.2%
8	14.0%	42.2%
7	16.4%	58.6%
6	13.9%	72.5%
5	11.0%	83.5%
4	8.2%	91.7%
3	5.6%	97.3%
2	2.7%	100.0%

From these, you can deduce that the chance of getting nine or more with two dice is around 28%.

Example: Monte-Carlo simulation of initial poker hand

What is the chance of getting a full house in the initial hand? To simulate this, you need to generate a large number of hands and randomly create the set of cards.

In the below solution, I generate a random number “ShuffleSeed” and order the deck by this number to get a shuffled deck. Then I deal ten hands with five cards each from the shuffled deck. The field “HandNo” is the ID for which hand it is.

This, I repeat 10000 times in a For – Next loop.

Finally, I analyze the result by additional Load statements using Group By, looking for pairs, three of a kind, full house etc.:

```
// ---- Create a Deck of cards
DeckOfCards:
Load   *,
    CardValue & ' of ' & Suit as CardName;
Load
    Pick(RecNo(),'Spades','Hearts','Diamonds','Clubs') as Suit,
    Pick(IterNo(),'2','3','4','5','6','7','8','9','10','Jack','Queen','King','Ace') as CardValue
    Autogenerate 4
        While IterNo() <= 13;
```

```
// ---- Shuffle and deal the deck many times
For vHandNo = 1 to 10000   // ---- ---- ---- ---- begin For-Next loop ---- ---- ---- ----
    // ---- Load the deck and assign a random number to each card
    LoadDeck:
    Load *,
        Rand() as ShuffleSeed
        Resident DeckOfCards;

    // ---- Order randomly and deal. Five consecutive cards form a hand
    PokerHands:
    Load
        CardName,
        Suit,
        CardValue,
        10*$(vHandNo) + Mod(RecNO(),10) as HandNo
        resident LoadDeck
            Where RecNo() <= 50
            Order By ShuffleSeed;

    Drop Table LoadDeck;
Next vHandNo   // ---- ---- ---- ---- end For-Next loop ---- ---- ---- ----
Drop Table DeckOfCards;

// ---- Check each hand for Flush
GroupByHandNo:
Load
    HandNo,
    If(Count(distinct Suit)=1,1,0) as HandHasAFlush
    Resident PokerHands
        Group By HandNo;

// ---- Check each card value in the hand for pair, three of a kind and four of a kind
GroupByHandNoAndCardValue:
Load
    HandNo,
    CardValue as CardInCombo,
    If(Count(CardName)=2,1,0) as ComboIsPair,
    If(Count(CardName)=3,1,0) as ComboIsThreeOfAKind,
    If(Count(CardName)=4,1,0) as ComboIsFourOfAKind
    Resident PokerHands
        Group By HandNo, CardValue;
```

```
// ---- Check each hand in the above table for two pairs and a full house
GroupByHandNo2:
Load *,
    If(HandHasAPair and HandHasThreeOfAKind, 1,0) as HandHasAFullHouse;
Load
    HandNo,
    If(Sum(ComboIsPair)=1,1,0) as HandHasAPair,
    If(Sum(ComboIsPair)=2,1,0) as HandHasTwoPairs,
    Max(ComboIsThreeOfAKind) as HandHasThreeOfAKind,
    Max(ComboIsFourOfAKind) as HandHasFourOfAKind
    Resident GroupByHandNoAndCardValue
        Group By HandNo;
```

The result is displayed in a pivot table with six expressions

Pair =	Count ({1<HandHasAPair={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)
Three of a Kind =	Count ({1< HandHasThreeOfAKind={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)
Four of a Kind =	Count ({1< HandHasFourOfAKind={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)
Two Pairs =	Count ({1< HandHasTwoPairs={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)
Full House =	Count ({1< HandHasAFullHouse={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)
Flush =	Count ({1< HandHasAFlush={1}>} DISTINCT HandNo) / Count ({1} DISTINCT HandNo)

Probabilities

Pair	42.0%
Three of a Kind	2.4%
Four of a Kind	0.01%
Two Pairs	4.8%
Full House	0.14%
Flush	0.22%

HIC