SANDIA REPORT

SAND2010-5107

Unlimited Release

Printed July 2010

The OVIS Analysis Architecture

J. M. Brandt, V. De Sapio, A. C. Gentile, J. R. Mayo, P. Pébay, D. C. Roe, D. Thompson, and
M. H. Wong

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation,
a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's
National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

The OVIS Analysis Architecture

J. M. Brandt

Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94551, U.S.A.
brandt@sandia.gov

A. C. Gentile

Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94551, U.S.A.
gentile@sandia.gov

P. Pébay

Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
pppebay@sandia.gov

D. Thompson

Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
dcthomp@sandia.gov

V. De Sapio

Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94551, U.S.A.
vdesap@sandia.gov

J. R. Mayo

Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
jmayo@sandia.gov

D. C. Roe

Sandia National Laboratories
M.S. 9292, P.O. Box 969
Livermore, CA 94551, U.S.A.
dcroe@sandia.gov

M. H. Wong

Sandia National Laboratories
M.S. 9152, P.O. Box 969
Livermore, CA 94551, U.S.A.
mhwong@sandia.gov

Abstract

This report summarizes the current statistical analysis capability of OVIS and how it works in conjunction with the OVIS data readers and interpolators. It also documents how to extend these capabilities.

Contents

1	Overview	5
1.1	Parallel Analysis Process	5
1.2	Local Analysis Process	7
2	Data Readers	11
2.1	The Base Class	11
2.2	Multiple Metrics On Components Of a Single Type	12
2.3	Future Directions	13
3	Interpolation	15
3.1	Per-component, piecewise-constant interpolation	15
3.2	Future directions	16
4	Observation Processing	19
5	Conclusion	21

1 Overview

OVIS [5] is a tool for parallel statistical analysis of sensor data to improve system reliability. Parallelism is achieved using a distributed data model: many sensors on similar components (metaphorically *sheep*) insert measurements into a series of databases on computers reserved for analyzing the measurements (metaphorically *shepherds*). Each shepherd node then processes the sheep data stored locally and the results are aggregated across all shepherds. OVIS uses the Visualization Tool Kit (VTK [1]) statistics algorithm class hierarchy to perform analysis of each process’s data but avoids VTK’s model aggregation stage which uses the Message Passing Interface (MPI); this is because if a single process in an MPI job fails, the entire job will fail. Instead, OVIS uses asynchronous database replication to aggregate statistical models.

OVIS has several additional features beyond those present in VTK that, first, accommodate its particular data format and, second, improve the memory and speed of the statistical analyses.

First, because many statistical algorithms are multivariate in nature and sensor data is typically univariate, interpolation of data is required to provide simultaneous observations of metrics. Note that in this report, we will refer to a single value obtained from a sensor as a *measurement* while a collection of multiple sensor values simultaneously present in the system is an *observation*. A base class for interpolation is provided that abstracts the operation of converting multiple sensor measurements into simultaneous observations. A concrete implementation is provided that performs piecewise constant temporal interpolation of multiple metrics across a single component.

Secondly, because calculations may summarize data too large to fit in memory OVIS analyses batches of observations at a time and aggregates these intermediate intra-process models as it goes before storing the final model for inter-process aggregation via database replication. This reduces the memory footprint of the analysis, interpolation, and the database client and server query processing. This also interleaves processing with the disk I/O required to fetch data from the database – also improving speed.

This report documents how OVIS performs analyses and how to create additional analysis components that fetch measurements from the database, perform interpolation, or perform operations on streamed observations (such as model updates or assessments). The rest of this section outlines the OVIS analysis algorithm and is followed by sections specific to each subtask. Note that we are limiting our discussion for now to the creation of a model from a set of measurements, and not including the assessment of observations using a model. The same framework can be used for assessment but that use case is not detailed in this report.

1.1 Parallel Analysis Process

An OVIS analysis begins when a request is inserted into the database of any shepherd node as shown in Figure 1α. A request consists of entries in several parameter tables and a single row in the `HaruspexRequests` table. A database trigger is used to signal the shepherd process on the same shepherd node as the database holding the new row whenever it is inserted into

Figure 1. The tasks involved in parallel data analysis using OVIS. Many tasks may run concurrently and execution times will vary, leading to different completion orders; however, any distribution task (β , δ_i) may fail and any shepherd participating the calculation will still have an approximate global model.

`HaruspexRequests`. Upon receiving this signal, each shepherd checks to see if the request origin is the local database or if it has been copied from another shepherd. If it is an original, the shepherd attempts to copy the request from its database to other shepherds' databases (Figure 1β). Regardless of whether the copy was attempted or was successful, the shepherd then performs the requested analysis (Figure 1γ_i) on its local sensor data. The result of this analysis is a model which is inserted into the local database. It is a local model since it only includes sensor data stored on one shepherd. In this way, all shepherds end up with several local models.

A second database trigger is used to signal the shepherd process when its local database has a new model inserted. If the new model is a local model and is specific to the data on the shepherd receiving the trigger, the shepherd copies the model into the databases of all other available shepherds (Figure 1δ_i). As long as the model which generated the signal is a local model (i.e., not global and regardless of whether the model is copied or whether the copy operation was successful), the shepherd then reads all of the local models from its database (whether they are local to the current shepherd's data or some other shepherd's data) and aggregates them into a single global model (Figure 1ε_i). The global model is inserted into the database.

In MPI parlance, computing the global model from local models is an all-reduce process. However, it should be clear that because tasks may complete at various times and individual shepherds may fail, several intermediate global models may be stored in the database before the final result is obtained. These intermediate models are approximations of the final global model. The law of large numbers informs us that as the number of local models contributing to the approximation increases, variations in the global model should diminish. OVIS relies on this behavior to provide information about the system in the face of failures, even in shepherds.

1.2 Local Analysis Process

Now that we have outlined a framework for performing statistical analysis in the face of possible failures, we present more detail on the local analysis process (Figure 1γ_i) in order to illustrate how it can be adapted to a wide range of calculations.

See Figure 2 for an illustration of all of the steps in the local analysis process. Only the “Learn” in Figure 2 operation is performed by a `vtkStatisticsAlgorithm` [2, 4]; the rest of the tasks shown in the figure are there to accommodate converting sensor measurements into observations and streaming data to and from the database.

The first task in any analysis is to convert raw measurements into observations. To support this, raw measurements must be read from the database (“Fetch subset” in Figure 2). Only a small number of measurements for each metric of interest are read at a time in order to reduce memory requirements. Interpolation (“Interpolate” in Figure 2) consumes measurements as observations are produced, so memory used to hold a subset of measurements may be refilled with more values from the database once all its entries are consumed. The abstract `ovHaruspexDataReader` is responsible for fetching and refilling `vtkTables` that hold measurements. These tables are then exposed to an `ovHaruspexInterpolator` subclass which produces observations from the measurements.

Figure 2. The tasks involved in local data analysis using OVIS. Each grey shape identifies tasks (arrows with mid-edge dots) that are repeated in a loop. Each rounded rectangle shows state that is used and modified by these tasks.

Whenever an `ovHaruspexInterpolator` subclass produces an observation, it is queued for analysis by calling an `ovHaruspexObservationProcessor` method. The cache of queued observations is periodically analysed using a `vtkStatisticsAlgorithm`.

The VTK statistics algorithms take in as input data `vtkTable` objects whose rows are observations and performs processing as shown in Figure 3. When the learn option is enabled, these observa-

Figure 3. The computations that a `vtkStatisticsAlgorithm` subclass can perform. The OVIS shepherd only employs the Learn phase.

tions are turned into a table of summary statistics – a model of the observations. If a pre-existing model is supplied, it is aggregated with the newly-learned model. When the derive option is enabled, additional statistics are calculated from the primary model; this new information can be more convenient or more familiar. When the assess option is enabled, each observation is marked

with one or more values corresponding to different assessments based on the model at hand. The shepherd only executes the learn phase of a statistics algorithm (“Learn” in Figure 2); derived quantities are computed on demand and only a minimal model is stored in the database.

As a set of queued observations reaches its maximum size, the `vtkStatisticsAlgorithm` is invoked and the resulting *intermediate model* is aggregated with the current approximation of the *local model* to obtain the next approximation of the local model. Finally, once all measurements have been converted into observations, the local approximation to the model is written to the local database by an instance of `ovHaruspexModelWriter`.

2 Data Readers

A data reader is responsible for presenting one or more tables of values to the interpolator in chunks that are small enough to fit into memory. Generally each table of values will be a subset of a metric table read from the database of the local shepherd but this is not a requirement; the tables may come from anywhere and have any format as long as the interpolator that is paired with the reader can accommodate it.

The default reader is intended for interpolating multiple attributes of a single component over time. That is, a single component may have measurements of multiple metrics such as temperature, fan speed, and so forth and the interpolator will approximate which values might be simultaneously observed on that component if the corresponding sensors all made measurements at one instant in time instead of staggered over time. In this case it is best to have measurements of each metric chronologically ordered so that interpolation may be performed by determining which table has the least recent measurement. It is important to note that any choice of a particular interpolation strategy implies that assumptions, at least implicit, be made regarding the nature of the data and how it evolves over time; also, different interpolation schemes can result in different sets of observations (e.g., piecewise constant as opposed to piecewise linear).

Other readers may provide data in formats that make other types of interpolation possible. For example, it may be useful to simultaneously observe one metric on some component and a different metric on a related component. As an example, consider observing power supply voltage for one node and the current drawn by the rack containing that node simultaneously. A reader for this type of analysis has not been implemented yet.

The following section will present the abstract base class. It is then followed by a section describing the current implementation and finally a discussion of features that may be desirable for future readers.

2.1 The Base Class

Any data reader must inherit `ovHaruspexDataReader`. This class owns a protected instance of the `ovHaruspexDataReaderPrivate` class that stores a vector of table names, `vtkTable` instances, and offsets used to track which chunk of the database each `vtkTable` currently holds. Subclasses may access this instance directly for their own use.

The base class also defines several virtual methods to be overridden by subclasses. The most significant are these pure virtual methods:

```
protected:
    virtual bool SetTable( vtkStdString, vtkTable* );
public:
    virtual vtkTable* ReadNextMetricValues( vtkIdType metricIndex );
```

When the reader is initialized with a list of tables to read from the database, it calls `SetTable()` and the subclass implementation is responsible for fetching the first chunk of data to fill the specified `vtkTable`. When the interpolator has finished processing one chunk of a particular table, it will call `ReadNextMetricValues()` to request the next chunk be read.

Note that reading chunks from metric tables is not straightforward because of conflicting requirements for (1) a unique index of the table, (2) chronologically ordered observations, and (3) fast constrained queries on metric tables. This is discussed more in the next section, which covers the details of the data reader subclass for fetching multiple metrics on components of the same type.

2.2 Multiple Metrics On Components Of a Single Type

A common use case for modeling and on-line assessment of cluster health is comparing the simultaneous values of multiple metrics on a single component to the same observations on other components of the same type. The `ovHaruspexComponentDataReader` class performs queries of this type.

In order to understand why this subclass functions the way it does, it is necessary to review the structure and usage of the metric data tables. Metrics are defined over component types; all components of a given type must be able to provide a single value for the metric at any point in time. We will also make the distinction that components of different types may have metrics with similar or identical names but they are not considered to be the same metric by OVIS. Each metric table has a column to store a unique integer (`TableKey`) serving as the primary key of the table, the component (`CompId`) over which a measurement is made, the time (`Time`) the measurement is made¹, and zero or more columns to store the metric value. The final column(s) are generally named `Value` but if the metric is a vector quantity other column names may be used. Sheep insert values into these tables, generally at regular intervals but use database-provided default values for the `TableKey` and `Time` columns. In this way, measurements over many components are serialized by their ordering with respect to `TableKey`.

This ordering is not required by the sheep but necessary for streaming chunks of tables through analyses efficiently. If the `TableKey` column were not present, the only way to obtain consecutive subsets of a metric table would be to execute multiple queries of the form

```
SELECT CompId,Value,Time FROM Metric...Values
WHERE ... (constraints to subset data) ...
ORDER BY Time,CompId LIMIT m OFFSET n
```

since without a unique index on each metric table there would be no other way to guarantee that multiple chunks are an exact partitioning of the desired subset of the table. Furthermore, because this query would need to be evaluated once for each chunk with different offsets and because each evaluation results in $m+n$ rows being fetched, the overall process of fetching chunks becomes quadratic in the size of the returned number of rows. This is unacceptable.

¹This is actually the time a metric value is recorded in the database in order to minimize clock skew.

The only solution is to provide an arbitrary but unique serialization of rows through a primary key. Because `TableKey` is arbitrary we do not wish to fetch it for all metric values – doing so would significantly increase the amount of memory and socket communication between the database and analysis processes without providing any benefit; instead we would prefer simply to use it in a constraint on each query that fetches chunks of a metric table.

2.3 Future Directions

The current reader fetches all measurements of a specified set of metrics on a specified list of components over a specified time interval. The measurements can thus be parameterized with 2 sets (metrics and components) and 2 times (the start and end of the interval). Another planned reader is one which takes a set of jobs and a set of metrics (defined over a single component type); this reader would report all metric measurements on components involved in the given jobs during the times when the job was scheduled – to characterize particular application codes – or alternately, metric measurements on the same components but only when they were not assigned any job – so that behavior during idle times can be characterized.

3 Interpolation

As discussed in §1.2, interpolation is the process of computing one or more individual measurements (on different components, at different times, or of different metrics) into a simultaneous (typically multivariate) observation. As an example, consider the measurements shown in Figure 4. Metric *A* will have 5 measurements reported in its chunk (3 values for component 1 and 2 for component 2). Metric *B* will have 6 measurements reported in its chunk (2 values for component 1 and 4 for component 2). The interpolator is responsible for taking these 2 tables and producing tuples simultaneously observed values of (A, B) . Let us assume we are interested in the case where values are considered to be simultaneous if they are evinced on the same component at the same time. In this case, because neither component has both metrics reported during the first second, no observation can be made until the next second. Note that interpolation must also cover cases where multiple measurements are made for a single component within a given interval (e.g., component 2 during the first second).

Figure 4. An example of how multiple metric measurements on multiple components might be reported to the interpolator; each metric has values chronologically ordered in chunks with values for multiple components interleaved.

Each interpolator is responsible for consuming measurements reported by its data reader and producing observations for subsequent processing. To this end, the `ovHaruspexInterpolator` base class has pointers to an `ovHaruspexDataReader` object and an `ovObservationProcessor` object. When a chunk of values has been consumed, the interpolator should ask the data reader for more values by calling the reader's `ReadNextMetricValues()` method. The interpolator invokes the `AddObservation()` method of the observation processor once for each observation it is able to generate. After all values from all chunks have been processed, the interpolator is responsible for calling `Finalize()` on the `ovObservationProcessor` in order that any cached observations may be processed.

3.1 Per-component, piecewise-constant interpolation

The current interpolator, `ovHarsupexInterpolatorSimple` outputs metrics observed contemporaneously on a given component. No more than a single observation will be generated for each second, so multiple reports of the same metric on the same component for the same second are

ignored; only the last reported value is used. This results in a piecewise constant time interpolation scheme, with the usual caveat attached to it: there is no attempt made to capture the dynamics of the observed metric between successive measure points. This is accomplished by maintaining a state table, which is now described.

Each row of the table corresponds to a component and there are columns for each metric value of interest as well as the component ID and observation time. The metric value columns are stored as `vtkVariantArray` objects so that values start out as invalid when no measurement of the metric has been made for a particular component. When each measurement is processed (in chronological order, across all metric chunks), its time is compared to the observation time of the corresponding row in the state table. If the measurement time is different and the row of the state table is complete (i.e., each metric column has a valid value), then an observation is generated using the state table values; the measurement value and updated time are not stored in the state table until *after* any observation is generated.

After all measurements have been processed, any rows in the state table that have been updated with measurements but for which no observation has yet been generated are visited in order to generate final observations before invoking `Finalize()` on the observation processor.

The `ovHaruspexInterpolatorSimple` class will process the measurements of our example data in Figure 4 as follows. Denote the measurements of metric *A* as $A_0, A_1, \dots, A_4$ and metric *B* as $B_0, B_1, \dots, B_5$. Say that measurement A_i is taken at time t_{A_i} and B_i at t_{B_i} with $0 \leq t_{A_0}, t_{B_1} < 1$, $1 \leq t_{A_1}, t_{A_2}, t_{B_2}, t_{B_3} < 2$, and $2 \leq t_{A_3}, t_{A_4}, t_{B_4}, t_{B_5} < 3$. Then the observation processor will be passed the following table, one row at a time:

Time	Component	<i>A</i>	<i>B</i>
1	2	A_1	B_3
1	1	A_2	B_2
2	1	A_3	B_5
2	2	A_4	B_4

Note that no values are reported for time 0 since no component has both metrics reported during that second. Also, values A_0 , B_0 , and B_1 are unreported since they are superseded by more recent measurements before a valid tuple has been constructed or before a second has elapsed.

3.2 Future directions

In addition to smoother time-interpolation (linear or higher-order), we have also planned a failure interpolator that would report a time interval with each observation indicating the amount of time an associated component functioned properly after the observation and whether a failure occurred at the end of that interval.

Finally, interpolation across associated components is desirable. In this case, one (or more) of the

specified associations between components would be used to pair measurements of different on different components into a single observation. For example, the CPU load of a compute node might be paired with the amount of network traffic through an associated switch port.

4 Observation Processing

When the interpolator creates an observation, it passes it to an observation processor. This processor, some concrete subclass of `ovHaruspexObservationProcessor`, is free to use the observations in any way it chooses. For example, one subclass named `ovHaruspexObservationAssessor` takes each observation and assessed it with respect to some previously computed statistical model to see how likely it is given the model. However, the subclass named `ovHaruspexModelUpdater` is what we are interested in; it aggregates the observations into a statistical model.

In theory, each new observation could be used to immediately update the statistical model. In practice, the model updater caches observations until the number of observations is large enough to efficiently perform a `vtkStatisticsAlgorithm` `learn` operation. This prevents memory exhaustion while limiting the overhead associated with aggregation.

Because the interpolator does nothing but convert measurements into observations, it is also possible to use the model updater to divide observations into groups that may be used to update multiple independent models. An example of this is the `ovHaruspexModelUpdaterMultiComp` class, which creates not a single cache of observations, but a cache per component. In this way, time-series analysis algorithms can be passed slices of component time histories that are large enough to recover trends in time.

5 Conclusion

The previous sections have documented each stage of processing measurements into a local statistical model, and how these stages are embodied in abstract classes with concrete implementations in subclasses. To conclude, we will discuss how concrete subclasses are instantiated by the shepherd in order to execute a particular analysis so that developers who wish to contribute extensions to OVIS can understand how to integrate new subclasses.

The shepherd runs each analysis in a separate thread. The thread function is the `ProcessRequest()` member of the `ovHaruspex` class. This function uses several other classes to help it perform work.

The `ovHaruspexReader` class is responsible for configuring the analysis by reading a request from a shepherd's local database. This includes reading

- the name of a `ovHaruspexDataReader` class to instantiate,
- the name of a `ovHaruspexInterpolator` class to instantiate,
- the name of a `ovHaruspexObservationProcessor` class to instantiate,
- the name of a `vtkStatisticsAlgorithm` class to instantiate,
- names and values of all string parameters stored in `HaruspexRequestsStringParameters`,
- names and values of all real parameters stored in `HaruspexRequestsDoubleParameters`,
- names and values of all integer parameters stored in `HaruspexRequestsIntParameters`,
- names and values of all component ids stored in `HaruspexRequestsComponents`, and
- names and values of all job ids stored in `HaruspexRequestsJobs`.

All of the parameters are passed to the `SetParameter()` method of each instantiated class, so any class must be able to accept unneeded parameters without complaint.

There is currently no mechanism to insure that the classes named in the request will work together; it is up to the baron or haruspex command-line tool to insert a semantically correct request into the database. So, if a new interpolator requires tables from a specific subclass of `ovHaruspexDataReader` this should be documented well and care taken in the GUI and CLI [3] to insert proper requests.

Once the classes have been instantiated and parameter values set, the `PrepareStorage()` method of the data reader is invoked to read an initial set of measurements. Then the interpolator's `Execute()` method is invoked. When it completes, `Finalize()` will have been called on the observation processor – which in our case must be a subclass of `ovHaruspexModelUpdater` so that it will have a local model ready for storage. The local model must always be a multiblock

dataset consisting of one or more tables. Currently, no other dataset types (such as `vtkGraph`) are supported.

At this point, `ovHaruspex` creates an `ovHaruspexModelWriter` instance which inserts the model tables into the database. This insertion causes a trigger to fire which eventually results in (1) the model being replicated to other shepherds as well as (2) the aggregation of all local models into a current estimate of the global model.

References

- [1] VTK Doxygen documentation. <http://www.vtk.org/doc/nightly/html>.
- [2] J. Bennett, P. Pébay, D. Roe, and D. Thompson. Numerically stable, single-pass, parallel statistics algorithms. In *Proc. 2009 IEEE International Conference on Cluster Computing*, New Orleans, LA, August 2009.
- [3] J. M. Brandt, A. C. Gentile, J. Mayo, P. Pébay, D. Thompson, and M. H. Wong. The Python command line interface to the OVIS analysis functionality. Sandia Report SAND2010-4289, Sandia National Laboratories, June 2010.
- [4] P. Pébay, D. Thompson, and J. Bennett. Computing contingency statistics in parallel: Design trade-offs and limiting cases. In *Proc. 2010 IEEE International Conference on Cluster Computing*, Heraklion, Crete, Greece, September 2010.
- [5] M. H. Wong, D. C. Thompson, P. Pébay, J. R. Mayo, A. C. Gentile, B. J. Debusschere, and J. M. Brandt. OVIS-2: A robust distributed architecture for scalable RAS. In *Proc. 22nd IEEE International Parallel & Distributed Processing Symposium*, Miami, FL, April 2008.

DISTRIBUTION:

1	MS 0807	Sudip Dosanjh, 1420
1	MS 0807	Bob Ballance, 9328
1	MS 9155	Robert L. Clay, 8953
2	MS 9152	Jim M. Brandt, 8953
2	MS 9152	Vince De Sapio, 8953
2	MS 9152	Ann C. Gentile, 8953
2	MS 9152	Jackson R. Mayo, 8953
2	MS 9051	Philippe P. Pébay, 8953
2	MS 9292	Diana C. Roe, 8623
2	MS 9152	David Thompson, 8953
2	MS 9152	Matthew Wong, 8953
2	MS 9018	Central Technical Files, 8945-1
2	MS 0899	Technical Library, 4536