A Robust Flight-Based Archival System for the Aircraft Situational Display To Industry (ASDI)

by

Micah E. Gutman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2001

© Micah E. Gutman, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 1 1 2001
LIBRARIES
BARKER

Author .....
Department of Electrical Engineering and Computer Science
December 15, 2000

Certified by .....
David K. Gifford
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Robust Flight-Based Archival System for the Aircraft Situational Display To Industry (ASDI)

by

Micah E. Gutman

Submitted to the Department of Electrical Engineering and Computer Science
on December 15, 2000, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

The aircraft situational display to industry (ASDI) is a commercially available data feed providing real-time information about the nation's air-traffic. This paper discusses a system capable of archiving the data in a database. Extra care was given to ensure to system robustness, enabling survival and recovery after the failure of any independent component. Scalability of the system was also crucial to the design, resulting in a database specifically designed to handle the tremendous amounts of data written over time. Particularly, the system needs to handle the archival of 60,000 flights per day, comprising 7.5 Mbytes of data. Instead of simply copying the feed into the database, a summarized record of each flight is constructed from numerous messages received from the feed over the lifetime of the flight. The creation of such flight summaries required a complex process to detect errors and ambiguities. The feed could not be treated as a "black box" that generated perfect information, but instead required careful analysis and modeling through a finite state machine. The resulting archival system is capable of resolving ambiguities and infer missing data from contextual information.

Thesis Supervisor: David K. Gifford
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Thanks to Professor Gifford, Jeanne Darling and Qian Wang for all their help.

Thanks to my family for all their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We have designed and implemented of an application that archives air traffic data from the aircraft situation display to industry (ASDI). Such a system poses numerous challenges that need to be addressed to make the system both useful and dependable. First, the system must be designed to handle large amounts of continuously streaming ASDI data. The storage of this information must be accounted for and measures must be taken to ensure its quick and easy access. Another difficulty arises from the fact that the real time information sent by the ASDI feed does not always translate easily into historical records. Ambiguities and missing data need to be accounted for to provide a more meaningful record of the nation's traffic. Lastly, the archival system must have safeguards that ensures continuous operation so that no data is lost. Redundancy must be part of the system design to protect against failure.

Our archival system that was built provides a flight-based history of the ASDI feed, summarizing all data about a flight in a single record. We employ a commercial database system as our foundation to provide a high level of performance and scalability for large amounts of data. Our archival system contains a number of modular components that can fail independently without the loss of any information. We studied the ASDI feed and built a finite state machine to model the different combinations of flight states and flight transitions that can be observed from the stream of information in the feed. Our finite state machine repairs inconsistent ASDI records, bringing consistency to the database. The archived data that results from this system

provides a valuable source of information that can be viewed on its own or used in other applications for presentation, analysis and data-mining.

Chapter 2

Motivation

Currently, the demand for air travel in the United States is facing increased demand and loads. According to the Federal Aviation Administration (FAA), the total system delays during 1994-97 were about 175,000 within the eight-month January-August period of each year. In 1998, this jumped to 221,701, and the data for 1999 show 265,226 delays for the same period. This represents a 50% rise in air travel delays. [2, 90] To make matters worse, air travel demand is projected to increase through the next decade. The FAA predicts that the 50.9 million flights this year will grow 25% to 63.9 million by 2010. [2, 90] Bottlenecks caused by airports nearing their capacities are of increasing concern in the air traffic control community. According to Dornheim [2, 90], "One major airline's study of the worst 10 days over the last 10 years showed that five of them occurred this spring."

As the system becomes more congested due to larger numbers of people traveling by airplane, it becomes more important to solve air traffic control problems. Dornheim [2, 91] explains, "One airline found that a 1-hour late aircraft affects an average of 3.27 other flights in the daytime, and 1.2 other flights toward the end of the day. It's the 1-hour or more late flights that cause the most damage, and these long delays are showing a disturbing rise." It is hoped that the analysis of historical data will offer new insights on how to alleviate this problem. Using the data collected by the archival system, applications can be developed that use analyze the data to make predictions ahead of time and clearing bottlenecks before they occur. The data can

offer insight into airport congestion, route usage and flight delays, providing the needed information to fix traffic problems.

Chapter 3

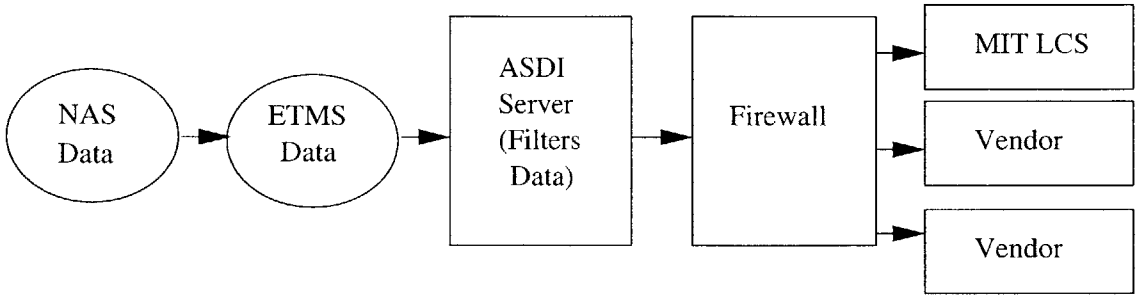
Existing Technology

3.1 The ASDI Feed

The ASDI feed provides continuous real-time air traffic data from the National Airspace System (NAS). The feed is sent to interested vendors via point-to-point links from the Volpe Center in Cambridge, Massachusetts, a cooperative project between the Federal Aviation Administration (FAA) and the Air Transport Association (ATA). The ASDI data originates from the FAA's Enhanced Traffic Management System (ETMS), supplying data about the United States and Canadian airspaces. Besides transmitting all NAS messages, ETMS processes flight data, producing additional messages that are sent over the feed. The ASDI server, situated at the Volpe Center, receives ETMS data and is responsible for the filtering of sensitive information that is deemed inappropriate for public use. Particularly, the Drug Enforcement Agency, military, and other government flights are considered restricted and information about them is not sent out for security purposes.

The filtering mechanism also removes certain types and parts of messages from the data stream for various security concerns. The data is then passed through a firewall to vendors and clients. The data arrives at MIT through a 128 kb/sec ISDN line. At the current time, the ASDI feed does not take up the full bandwidth of the line. In fact, less than half of the bandwidth is actually used. However, the FAA has plans to increase the amount of information sent over the feed, sending more frequent position

Figure 3-1: The path of the ASDI Data Stream



updates on each plane, necessitating full use of the ISDN bandwidth. A summary of the data flow through this feed is shown in Figure 3-1.

The data sent through the ASDI feed consists of ASCII based messages. Various message types are defined, each representing a different "flight event", producing a different message formatting for each type of data record. Additionally, each message transmitted through the ASDI feed is framed with information that is common to all. Particularly, each message has a unique sequence number, a time stamp and a facility identifier (denoting the origin of the message). Two types of messages are sent over the data feed, original messages from the NAS tracking system and ETMS generated messages. The NAS messages include those for aircraft arrivals, departures, flight plan information, cancellations, flight plan amendments, position updates, and boundary crossings. The ETMS generated messages incorporate additional calculations based upon the NAS messages. For example, ETMS messages offer more detailed route information resolving airways and fixes into latitude and longitude coordinates. Because the data in the ETMS data can be calculated from the NAS messages, it is not archived by the system to reduce storage requirements. Thus, the archival system is only concerned with the seven NAS messages whose contents are summarized in table 3.1.

Table 3.1: ASDI Messages and Contents

Message Type	Description	Data in Message
AF	Flight Plan Amendment	Aircraft Identifier, Origin, Destination, Amendment Data
AZ	Arrival Announcement	Aircraft Identifier, Origin, Destination, Arrival Time
DZ	Departure Announcement	Aircraft Identifier, Origin, Destination, Departure Time, Estimated Arrival Time
FZ	Flight Plan	Aircraft Identifier, Assigned Speed, Assigned Altitude, Route, Aircraft Type, Proposed Departure Time
RZ	Cancellation	Aircraft Identifier, Origin, Destination
TZ	Position Update	Aircraft Identifier, Altitude, Coordinates (latitude/longitude)
UZ	Flight Plan Update	Aircraft Identifier, Speed, Altitude, Boundary Crossing Time, Aircraft Type

3.2 Other ASDI Applications

Before building the archival system, other applications using the ASDI feed were examined. Dimension International's FlightExplorer [4] and thetrip.com's FlightTracker [3] were examined as sample applications that currently allow a user to access the ASDI data aircraft data. Both offer presentation of real time data, with limited amounts of historical data. In general, the applications only store information pertaining to airborne flights. Once a flight is completed, the applications remove all information about the flight from memory. For the most part, the same information can be obtained from either program, although their respective presentation style and interfaces are different. Using the applications, the positions, speed, altitude, and plane type can be found for any flight.

FlightExplorer presents a zoomable geographic map, containing the locations of all aircraft. The positions of the planes are continuously updated, giving the user a sense of airplane movement, density and positions. FlightExplorer also contains selection features that allow the user to limit the displayed planes to a given set of search criteria. The program provides a broad set of querying fields, ranging from

destination, altitudes and plane types (i.e. 747's). The professional version of this software also incorporates weather information, overlaying weather patterns on the map alongside the planes.

In comparison, FlightTracker has a much less involved interface. The user enters a flight or airport for which he would like information. The program responds to the query by mapping out the airplane's position on a map, providing all relevant information about the flight in an accompanying window. The interface provided by FlightExplorer clearly makes more information available to the user. The visualization of all flights at once provides the user with a sense of how many planes are in the air and the air traffic density, information that is not as easily available with FlightTracker. The querying provided by FlightExplorer is broader, giving the user more flexibility in limiting the planes in which he is interested. Apparently, no existing applications offer ASDI data on flights that have already landed. In this way, the archiving system is the first of its kind, giving a complete historical account of IFR air traffic.

Chapter 4

System Requirements

A number of requirements were imposed on the design of the ASDI archival system:

1) The system has to be robust and facilitate redundancy. It was noted that a system archiving the nation's air traffic must be in continuous operation. Any down-time would result in a loss of irreplaceable historical data. Since the data collected by the system would ultimately be used for research and analysis, it was important to prevent gaps in the historic record to provide a reliable source of information. To ensure this, it was important for the system to have the ability to recover easily when one of its components fails. Additionally, the system would need to continue running during outages using fallback processes that would continue recording data until full recovery was attained.

2) The system has to be scalable. A large amount of data will be collected by the system and it will continuously grow. This property poses two distinct challenges. First, the system needs an efficient and scalable method for storage of the data. The information stored by the system needs to be streamlined and compressed. Additionally, a strategy is needed to manage the physical storage of the system so new data can always be written. A second challenge posed by the large amount of data is assuring quick access to individual records. It is important that the system be designed to be usable. Queries need to be computed quickly so the system does not become bogged down by the size of its historical database.

3) The system needs to convert data presented in a real time format to a historical format. The ASDI feed is an event based real time feed. Data pertaining to a specific flight is thus fragmented into many smaller messages detailing a "flight event" (e.g. departure, position update, arrival). Users and applications that access the historical data would prefer a consolidated version of the data, one in which it was unnecessary to piece a flight's information together from individual messages. Additionally, it is undesirable to save the ASDI feed in the real time format since messages often repeat information, wasting valuable storage space. These two factors made it necessary to design a system that consolidates fragmented real time information into compact historical flight records.

4) The system has to function with an ASDI stream of unknown reliability and sometimes uncertain behavior. At the start of the project, it was unknown whether the ASDI feed sent out complete and correct data, and indeed it was safer not to assume so. Thus, it was necessary for the system to deal with the presence of unexpected data, and the absence of expected data when constructing the history of a flight from the real-time messages. Without such measures, the system would lose data in situations it could not handle. Of course, the flip side of this requirement must be also considered. Feed data that was incomprehensible and did not give adequate information about a flight should not be saved. A balance had to be found where the system was neither too rigid nor too flexible in its interpretation of the data feed.

5) Message consolidation and entry into storage has to occur at least as fast as the data rate. The ASDI feed continuously delivers new messages to the system. Any system seeking to archive the data needs to work at least as fast as the speed at which data arrives. If this was not the case, the system would eventually break from the backlog of stored messages. Note that this requirement pertains to the average data and processing rates. It would be acceptable for the system to fall behind in processing if it was able to catch up at some later time.

6) The system has to infer contextual information and make calculations not given explicitly by the data feed. All information needed for an accurate

historical record is not contained explicitly within the feed. Some information had to be inferred by the system and added to the historical record. It was important that inferred information be carefully generated so as not to introduce errors into the historical data.

Chapter 5

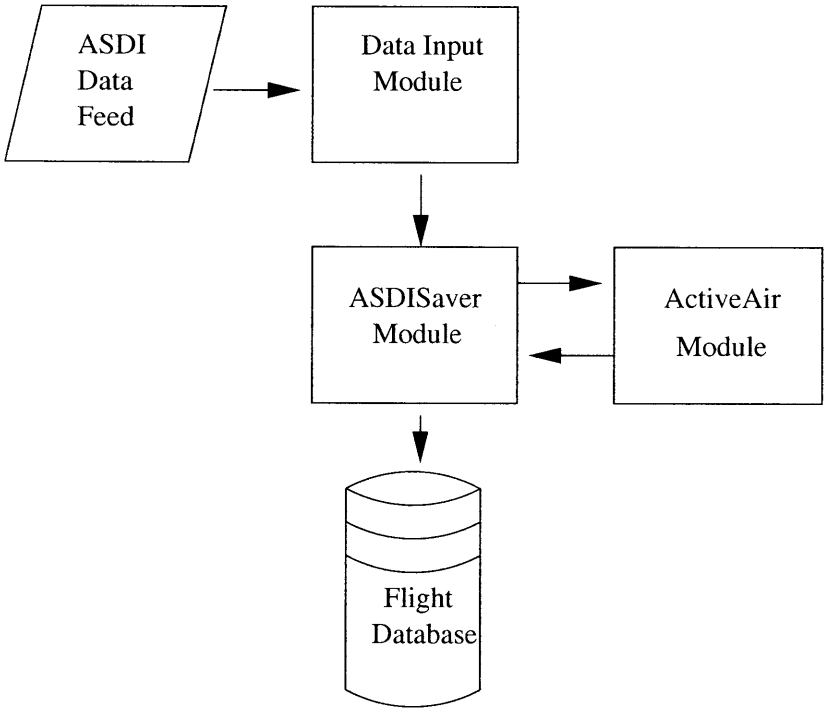
System Design

5.1 Overview

The ASDI archival system can be broken up into 4 main modules: data input, the ActiveAir table, the flight writer, and the flight database. The data input module deals with the incoming data to the system. It retrieves messages from the ASDI feed and parses it for use by the system. The activeair table is responsible for the conversion of the real time data into a historical format. The table collects all information about active flights, compacting it into a single historical record. The flight writer module is responsible for the entry of completed historical records into the database. The flight database module serves as the storage unit for the flight data. The relationship between the various modules is illustrated in figure 5-1.

All code was written in Java version 1.2, allowing the software to run on any platform. Interaction with the database was carried out with the JDBC. The entire system was designed to run on a minimum of two machines, a dedicated database server, and at least one machine running the ASDISaver module. The ASDISaver machine requirements are rather small. The ASDISaver application was successfully run on a Pentium 120 processor with 96 MB of RAM.

Figure 5-1: Module Interaction and Data Flow



5.2 Data Input Module

By far the simplest component, the data input module is responsible for retrieving data from the ASDI feed, making it available for the rest of the system. Data is obtained from a server connected to the ISDN line that allows client connections for the retrieval of data. The server implements the necessary security to ensure that unauthorized users do not obtain access to the raw ASDI feed.

In making its connection to the ASDI server, the data module implements a timer that times out the connection if no data is received within a sixty second time period. The timing out of the connection is used to alert the ASDISaver of a dead connection so it can attempt to reconnect. (See section 5.5.2)

Once data is obtained from the ASDI feed, it is parsed. Each text-based message is converted into a data structure in memory that allows for easy access of the message fields. Some message fields are sent in an encoded format using a smaller character set. The parser carries out the necessary decoding so the fields are easily understood. The parsing of each message is specific to the message type, since each contains different fields, and formats representing different types of information often generated by different systems.

The entire data module was designed to work as an independent component that can be reused by other applications that require the use of the ASDI feed. Indeed, the module performs an important task that is necessary by most applications that attempt to read data from the ASDI feed. The module abstracts the process of accessing the text-based data feed, returning a Java representation of each message. Although simple in operation and complexity, the module still comprises a fair amount of coding that embodies the necessary knowledge on how to parse and decode the various messages that are sent over the feed.

5.3 Database

5.3.1 Hardware Design and Structure

The most important component of the system, the database serves as long term storage for the air traffic information collected from the feed. The system uses an Oracle 8i v8.1.5 database running on Red Hat Linux v6.2. Oracle was installed on a dedicated server, containing dual Pentium Pro 200 processors and 256 megabytes of RAM. Two hard drives were installed in the machine. A 3 gigabyte hard drive serves as the machine's main operating storage. This drive contains the operating system, Oracle software, and the system tablespace definitions for the database. The system tablespace provides Oracle with information about the data structures that store the data. A second 17 gigabyte hard drive was installed solely for the storage of air traffic data. All operations accessing the data use the system tablespace definitions to locate and access the data. By separating the system data and the air traffic data on two separate disks, database operations can concurrently use both disk drives and processors all at the same time.

5.3.2 Data Model

The ASDI feed sends over 2 million messages over a typical day, providing information about over 50,000 flights. These messages are event oriented, announcing flight information pertaining to specific real time events (i.e., departures, arrival, position updates). Although this format works well for real time applications, it is not optimal for historical analysis. Users of historical data would most likely be interested in a flight's full history, which is spread out over multiple messages. Thus, the database uses a flight oriented data model, storing the full data for each flight in a table row, instead of storing each individual message in a row. This format gives users a clear and concise history of an entire flight, without having to piece together multiple messages.

Storing the data in flight records introduces a number of other benefits as well, helping reduce the database size and making it more manageable. A series of related

messages contain redundant information since some flight information is repeated across many of these messages. Fields such as flight number, departure point, destination, and aircraft type are transmitted multiple times over the course of a flight. Storing the ASDI feed by flight reduces the overall number of rows in the table from 2 million rows to 50,000 rows per day. The reduced number of rows allows faster querying and more efficient usage of physical storage space. Lastly, the flight format reduces the number of write operations necessary by the archival system. Writing each message to the database would expend a good portion of the database's time. The performance of other database operations, such as queries from users and applications would suffer as a result. Writing the full history of a flight out at once significantly reduces the burden on the database.

The downside of not using a message based data model comes from having to convert the original feed into flight records. The system requires a module that tracks ASDI messages and generates the individual flights. (See section 5.4) Although conceptually simple, the process of linking together messages is technically complex and many problems arise. This conversion adds extra failure points into the system, and opens up the possibility for error in the stored data. Although such errors can be avoided entirely by storing the data by messages instead, doing so simply shifts the burden to the user who is forced to combine messages together himself, who in turn can make the same mistakes.

All flight rows are stored in a single table, named FlightTable. The columns of FlightTable are described in table 5.1. Many of the table rows allow null values. In these cases, a value is not always available for insertion, and space is conserved by using a null rather than a "filler" value. Note that every column requires an aircraft identifier, origin, destination, departure time, and arrival time. The not null constraint on these columns ensures a minimal means of identification for every flight. Three other not null columns, proposedfirst, deptimevalid, and actualdeptime valid, help describe the ASDI messages that generated the row, giving an indication of the reliability of the data. (See section 5.4.8)

The expected sizes were calculated from a week of data. The database uses a 7 bit

Table 5.1: FlightTable Columns

Column Name	Description	Type	Allows Null	Size, mean +/- sd (Bytes)
aircraftid	The flight number or tail number	varchar2(11)	no	7
deppoint	Departure point	varchar2(12)	no	4
dest	Destination	varchar2(12)	no	4
origdest	Original destination, if it changed in the air	varchar2(12)	yes	4
originaldeptime	The planned original departure time	date	yes	7
actualdeptime	The time the plane actually departed	date	no	7
deptimevalid	Departure message received for flight	number(1)	no	1
etafromdep	Eta at the time of departure	date	yes	7
arrtime	Arrival time	date	no	7
arrtimevalid	Arrival message received for flight	number(1)	no	1
numberaircraft	Number of aircraft	number(2)	yes	1
heavyindicator	Heavy indicator	varchar2(1)	yes	1
aircrafttype	Aircraft type	varchar2(4)	yes	4
equipmentqualifier	Equipment Qualifier	varchar2(1)	yes	1
assignedspeed	Assigned Speed	varchar2(4)	yes	4
assignedaltitude	Assigned Altitude	varchar2(7)	yes	7
highspeed	Highest observed speed	number(4)	yes	4
highaltitude	Highest altitude observed	number(4)	yes	4
avgspeed	Average observed speed	number(4)	yes	4
avgaltitude	Average observed altitude	number(4)	yes	4
proposedfirst	Whether proposed flight plan received for flight	number(1)	no	1
route	Route	varchar2(4000)	yes	33
timeentered	Time record written to database.	date	no	7

ASCII character set for all of the varchar2 columns (textual data), thus each character takes up only one byte. The varchar2 and number columns vary in size taking up only the amount of space used by the data, up to the maximum specified in its type definition. The route column in particular takes advantage of this characteristic, allowing values up to 4000 bytes in size while averaging only 32 bytes. The extra space allows for the entry of more sophisticated route information that is available over the feed. The larger column size makes it easier for this information to be included in the future, if desired. The aircraftid, origin, and destination allow more characters than expected to conform with the ASDI feed specifications that allow larger but seldom seen values.

5.3.3 Improving Query Performance

Even after converting messages to full flight records, we estimate that 2.5 gigabytes will be required for a year of database records. It is crucial that measures be taken to ensure that the database's size does not negatively affect the speed of performed operations on the FlightTable. This problem is addressed through table partitioning and indexing.

Table Partitioning

The first measure we take to manage the size of the database is the partitioning of the FlightTable. Partitioning is a tool used to split a table up into smaller pieces. In this case, the FlightTable is partitioned by range using the actual departure time column as the partition key. The table contains a partition for every month, with all planes departing within a given month appearing in the partition. Partitioning offers additional functionality as well, allowing the user to query and manipulate data on specific partitions. Additionally, each partition can be manipulated as a whole, being added or deleted to the greater table as a unit, allowing for the easy movement of table data.

The greatest benefits come from the transparency of the underlying partition

structure when performing table-level operations, however. Table inserts automatically place new rows in the correct partition based upon the key. Similarly, queries on the table automatically search appropriate partitions, returning results more quickly than if it considered the full table. Thus, any query limiting the departure date can efficiently trim its search space using the partitions. For example,

*select * from flighttable where actualdeptime ≥ '20000101 000000';*

automatically chops off any partitions before January 1, 2000, returning all the columns for any flight after that date and avoiding the lengthy process of examining the actualdeptime field for each row in the table (or index if one exists.) In effect, limiting on the partition key simply determines the time frame of the query. This fact makes it a good idea to include the departure time whenever possible to cut back on the amount of time it takes to return results. For example,

select avg(avgspeed) from flighttable where aircrafttype = 'B747';

requires searching on all partitions for the average of the average trip speed for all Boeing 747's. However, the same model of plane probably exhibits little variation in its average speed every month. Thus, a faster query giving a similar result would be,

select avg(avgspeed) from flighttable where aircrafttype = 'B747' and actualdeptime ≥ '20000101 000000' and actualdeptime < '20000102 000000';

which limits the results to data collected over a one month period of time. A special case arises for queries that deal with one of the other three time columns (original departure time, estimated time of arrival at departure, arrival time). These queries should always include a clause limiting the actual departure time since it is already implied elsewhere in the statement. For example:

*select * from flighttable where arrtime ≥ '20000101 000000';*

returns all the columns for any flight which arrived after January 1, 2000. Any plane arriving after this date could not have departed too long before this date. Thus,

*select * from flighttable where arrtime ≥ '20000101 000000' and actualdeptime ≥ '19991231 000000';*

should return the same data as the previous query since any plane landing after January 1, 2000 would have departed by December 31, 1999. The second query has

the advantage of utilizing the partitioning key, and users and applications should always include the actual departure time in all queries that limit the results on any of the time columns. Queries containing the estimated time of arrival column should similarly search for flights departing one day previous, while original departure time queries should search for flights departing one day after.

Table Indexing

A second tool for improving query performance in the FlightTable is the use of indices. An index is a data structure stored in the database that gives quick access to information in the table. An index is associated with a key that is related to the columns of the underlying table. Queries can use the index as a lookup table, quickly locating rows containing the column value using the index key. Use of an index involves a tradeoff though, taking up memory to store the data structure and decreasing performance for inserts, updates and deletes which now have to modify the index as well when the underlying table is modified. Thus, indices were only created on columns that had high-anticipated use as a restricting condition in queries.

With this tradeoff in mind, five indices were created on the FlightTable. They are described in table 5.2. The five indexes were created to allow quick independent access for queries based on five attributes, flight time, origin, destination, flight number/tail number, and plane type. Although only the actual departure time is indexed, users can still take advantage of the FlightTime for one of the other three "time" column with the same procedure used to take advantage of the table-partitioning key. (see section 5.3.3) A clause restricting the actual departure time can always be added to any statement considering any of the other three times. All of the indexes have complex keys, made up of more than one column. The additional key columns further order the index, providing an ordering of rows that have equivalent values for the first key column. For example, two flights with the same destination would both be found under its destination in the FlightDest index, with the earlier flight appearing first due to the secondary key. The FlightTime index contains four columns in its key. This is related to the unique constraint on the index and is discussed further in

Table 5.2: FlightTable Indices

Index Name	Index Keys (in order of precedence)	Attributes
FlightTime	actualdeptime, aircraftid, deppoint, dest	unique, local
FlightDest	dest, actualdeptime	local
FlightDepPoint	deppoint, actualdeptime	local
FlightId	aircraftid, actualdeptime	local
FlightType	aircrafttype, actualdeptime	local

section 5.3.5. Lastly, all the indices are local, meaning a separate index is built for each table partition. Oracle does not allow global indexes that use a different column than the partition key. As a result, Oracle needs to search through the indices of all appropriate partitions when retrieving results.

5.3.4 Physical Storage and Scalability

The database size also poses challenges for physical storage. Since the feed is continuously running, the database is always in a state of growth. Although a large hard drive can store a few years worth of data, a plan is needed to prevent the system from reaching its capacity and breaking.

After collecting data for a few weeks, the observed average size of a table row can be approximated at 125 bytes of data. (See table 5.1). Using an upper bound of 60,000 flights per day, a month's worth of data would take up 213,125,000 bytes of disk space, approximately 2.5 gigabytes per year. This figure does not include the overhead used by Oracle. Even with generous allowances for this overhead, the 17-gigabyte hard drive used by the database can still hold a few years worth of data. One long-term option for the system is the installation of additional hard drives when current storage becomes full. Since memory devices are becoming cheaper and larger, future disks would be able to hold a larger amount of data, requiring less frequent replacement. It is conceivably possible that there will be space for hundreds of years of data in the time it takes for the system to fill up only a few hard drives.

A second means of ensuring scalability is by transferring data to a cheaper storage

device, such as a CD. The database is already divided into partitions, providing an easy way to divide the database into pieces. Taking up a little more than 200 MB, a month's worth of data can easily fit onto a CD even after overhead is taken into account. Once copied to the CD, table partitions can be deleted from the hard drive if desired, freeing up space for future data. The same partition can also be added back from the CD at a later point if the data is needed. The option also offers two additional benefits as a side effect. First, the CD's provide a means of backup for the database. If the database is ever lost, it can be rebuilt from the CD's. Second, the CD's give an easy means of distribution of the ASDI data. Parties interested in this can be given a copy of the appropriate CD's. Thus, it would seem desirable to transfer data onto CD's regardless of whether or not the first option of adding new hard drives is used.

5.3.5 Flight Uniqueness

It is important that each flight get entered into the database a single time as a unique row. If this were not the case, there would be ambiguity with multiple rows describing the same flight. Additionally, disk space would be wasted by the duplicate rows, increasing the burden on both query performance and physical storage. Clearly, some measure needs to be taken to ensure the uniqueness of each flight. The database contains two distinct methods of ensuring uniqueness.

The common method of guaranteeing uniqueness in a relational database is through a column constraint. Once in place, an attempt to insert a non-unique row results in an error message and abortion of the operation. Oracle allows table columns to be specified as unique ensuring that each row contains a different value. The uniqueness constraint can be placed on combinations of columns as well, ensuring that the data contained in the full combination is different, allowing a value in an individual column to be repeated. Oracle maintains the uniqueness constraint through an index, hence the unique attribute of the FlightTime index (see section 5.3.3). The FlightTime index contains four columns as its key. Of these four, at least one must be different between any two rows stored in the table. Note that it is necessary to use

multiple columns for uniqueness since any one column cannot identify a flight on its own. A particular flight with the same flight number or tail number can fly the same route repeatedly over time, creating numerous records in the database. Similarly, multiple planes with different flight number can depart at the same time. However, a particular flight can only make a single departure at a time, providing the means to fully identify a flight. Note that the key also includes the departure point and origin. These two additional columns are included for the query performance role of the index and are not needed for the uniqueness constraint.

The initial system design called for the storage of rows containing only perfect data. Under this assumption, flights for which data is missing would be excluded from the database. A later redesign relaxed this requirement, allowing estimates to be used in places where data was missing. (See section 5.4 for further details). Because rows are no longer perfect, it is possible that multiple independent data writers would attempt to write different information for the same flight into the database. The previous constraint ensured that multiple parties writing to the database had the exact same data about each flight, which is now no longer the case. One writer may have received data that the other one did not, or it is possible that the two made different estimates. This is not a trivial case to consider, and there is a good chance of this situation occurring since the system was specifically designed to allow multiple writers for the sake of redundancy. (See section 5.5.2) This situation creates two problems. First, if multiple writers have different values for the departure time, the unique key would be bypassed and the table would end up containing multiple rows for the same flight. Second, the unique constraint favors the writer who writes first. Although a subsequent writer may have genuine data or better estimates for some columns, its insert operation could possibly be aborted by the unique constraint simply because a row already exists.

Clearly, the database needs a more intelligent insertion process that can better identify flights. The process would also need the ability to update already existing rows, combining information from multiple sources when necessary. To do this, a stored procedure, enterNewFlight was written and entered into Oracle. Instead of

performing inserts into the table directly, the stored procedure is called instead with new data and ensures the uniqueness of each flight, intelligently combining new data with existing rows.

The enterNewFlight procedure goes through a multiple tiered process when determining whether a flight exists. The procedure starts by counting the number of flights in the database with the same aircraft identifier, departure point and destination that took off 12 hours before and after the departure time of the new flight. It is assumed that any previous estimates made about this flight would fall within this range. If no matching flight exists in this window, it is presumed safe to enter this record into the database. Next, the procedure checks if there are any flights in the database that have the exact same information as the new flight. If a row with the same data exists, the procedure immediately exits, doing nothing with the new flight record.

These first two steps take care of the most common cases and allow the procedure to act efficiently in its computations. Most aircraft do not fly between the same origin and destination within a twenty four hour span. Thus, most new flights would be inserted in this first step of the procedure, without having to go through additional computation. Only flights making the same trip in a twenty four hour time span would need to go beyond this step to ensure that they are distinct. The second step deals with cases when the information from two sources is exactly alike. Despite the concern for differences, most of the time multiple data writers read from the exact same feed and attempt to write identical flight records. This second step immediately halts computation for this common case, identifying this most common type of duplicate.

The rest of the procedure now deals with the two less common cases, the insertion of a flight between two points that was made more than once in a twenty four hour time span, and the case when writers have different information that needs to be combined (or ignored) to form a single record. The enterNewFlight procedure first retrieves all flights in the twenty four hour window with the same aircraft identifier, origin and destination. It then examines each one to determine whether the new flight overlaps the time that the pre-existing flight was in the air. If no overlap is

found, the procedure concludes that the flight is different than the one already in the database and that the trip was simply made more than once in the same twenty four hour period. In this case, the flight record is inserted into the database as normal. If an overlap is found though, the procedure updates the existing database row with pertinent information from the new flight data. Specifically, it checks whether the new data contains genuine times, replacing any estimates that were previously placed in the database. If both records have estimated data, it uses the better estimate, entering the later arrival and departure times. (The error made in the estimate makes the times earlier than they should be. See section 5.4.8) Finally, if the old row was missing flight plan information prior to takeoff, the fields are filled in with information from the new flight record if available.

Although the procedure does a good job of matching new flights to pre-existing flights, it is still theoretically possible for two rows to be inserted describing the same flight. This case is best exhibited by an example. Let writer-1 track flight 123 from 1:00 until 2:00, after which it stops receiving any information about the flight. Writer-1 will then send a record to the database with its estimate that the flight occurred between 1:00 and 2:00. Let writer-2 receive information about the same flight 123, starting at 3:00 and ending at 5:00. The enterNewFlight procedure would treat the record submitted by writer-2 incorrectly, as a different flight than the one submitted by writer-1.

This error is indicative of a tradeoff that had to be made. The system database needs to balance the possibility of duplicate entries for the same flight with the possibility of accidentally combining two distinct flights into one record. In this case, the system prefers to err on the side of duplicate flights. Whereas information is lost when two distinct flights are combined, the user still has the ability to interpret duplicate records on his own when they are put into the database. Additionally, the occurrence of an aircraft making multiple trips between two close cities is common. Regular shuttle flights often repeat a trip of short duration numerous times in a day. Extra care had to be taken to avoid merging two adjacent flights. The probability of getting duplicate flights is still relatively small though since it would require writers

to come on and off-line in a specific unusual order. Since the writers are not expected to go down very often, it is unlikely that flights would be duplicated in the database too often.

5.4 ActiveAirModule

5.4.1 Finite State Machine

The main purpose of the ActiveAir module is to collect messages over the life span of a flight, piecing them together into a single record. Because the feed is meant to be a tool for real time analysis, the sequence of messages received for a given flight does not always translate easily when forming historical records. It was necessary to develop procedures for dealing with varying ASDI feed behaviors. An organized process had to be developed to track the history of a flight, using the real-time glimpses that were conveyed over the ASDI feed and filling in gaps of a flight's history.

The different messages received about a flight convey information about the state of an airplane. The state is an important indicator for the flight, affecting the module's treatment of future incoming messages. To facilitate this, a finite state machine (FSM) was constructed, defining possible plane states and transitions. Upon the arrival of a new message, a transition is inputted into the state machine, which then returns the new state of the plane. The FSM also alerts the ActiveAir module of illegal transitions, identifying unexpected behaviors in the feed that could indicate missing messages. The FSM is implemented as a stand-alone sub module, which can be used separately from the ActiveAir module.

The state machine uses six unique plane states, pending, in-air, cancelled, cancelled-in-air, completed, and unknown. The pending state represents a flight that has not taken off yet, for which a proposed flight plan has been received. The in-air state corresponds to flights that are currently flying. The cancelled state represents flights that are cancelled while the plane is still on the ground before takeoff while the cancelled-in-air state corresponds to flights cancelled after takeoff in mid-flight. The completed

state is used for flights that land at their destination. The unknown state is a special state, used to initialize new flights and deal with the first transition, representing the starting point of the FSM.

It was necessary to distinguish between the two types of cancellations to handle the feed behavior of cancellations immediately followed by reinstatements with a new flight plan. As an alternative to sending amendment (AF) messages to change the flight plan, a sequence of cancellation (RZ) and flight plan (FZ) messages can be sent instead. Upon reinstatement of the cancelled flight, it is necessary to return the plane to the same state it was in prior to cancellation. The use of two distinct cancellation states allows the FSM to distinguish between cancellations occurring before and after departure, returning the plane to the correct state upon reinstatement. The cancelled-in-air state also has an important role in sending stale records to the database in conjunction with garbage collection. (See section 5.4.6) Whereas flights that are left in the cancelled state are deleted entirely since they never occurred, flights that are cancelled-in-air are placed into the database since the flight actually did take place and was simply removed from the tracking system at some point in mid-flight.

The state machine recognizes five transitions, update, takeoff, position-update, cancellation, and landing. An update represents the arrival of a message updating the flight plan of the plane. On the other hand, position-updates correspond to messages conveying the actual position and speed of the plane. Takeoff, landing and cancellation transitions represent a flights departure, arrival and cancellation, respectively.

Every message type sent over the ASDI feed is paired with one of the transitions. Upon the arrival of each message, the implied transition is taken in the flights FSM to realize the flight's new state. Flight plan (FZ) and amendment (AF) messages correspond to update transitions. Position update (TZ) and boundary crossing (UZ) messages corresponds to the position-update transition. Note that besides announcing a boundary crossing, the UZ message also contains flight plan information similar to the data contained in FZ messages. However, UZ (and TZ) messages can only be received while the plane is in the air while FZ and AF messages may be received in-

air or on the ground. Thus, the UZ message is classified as a position update. As their names suggest, departure (DZ), arrival (AZ), and cancellation (RZ) messages refer to takeoff, landing and cancellation transitions respectively. Also, FZ and AF messages may trigger a takeoff transition, as well. Both of these messages allow the time field to correspond to a departure time. When this is detected, the takeoff transition is inputted into the finite state machine.

The transitions allowed by the FSM are listed in table 5.3. Normally, a plane would start in the pending state after the first flight plan message is received in a FZ. Additional flight plan updates may be received prior to takeoff, until the takeoff transition moves the plane into the in-air state. Once in the air, the plane can undergo both update and position-update transitions as needed. A landing transition then eventually completes the flight. Additionally, it would be considered normal for flights to enter the cancelled and cancelled-in-air states with a cancellation transition and then become reinstated with a new update transition. The normal path a flight takes through the FSM is exhibited in figure 5-2. Although most flights follow this pattern, unfortunately a significant number exhibit degenerate behavior, forcing the inclusion of additional transitions, described in the next section.

5.4.2 Explanation of Non-Obvious Transitions

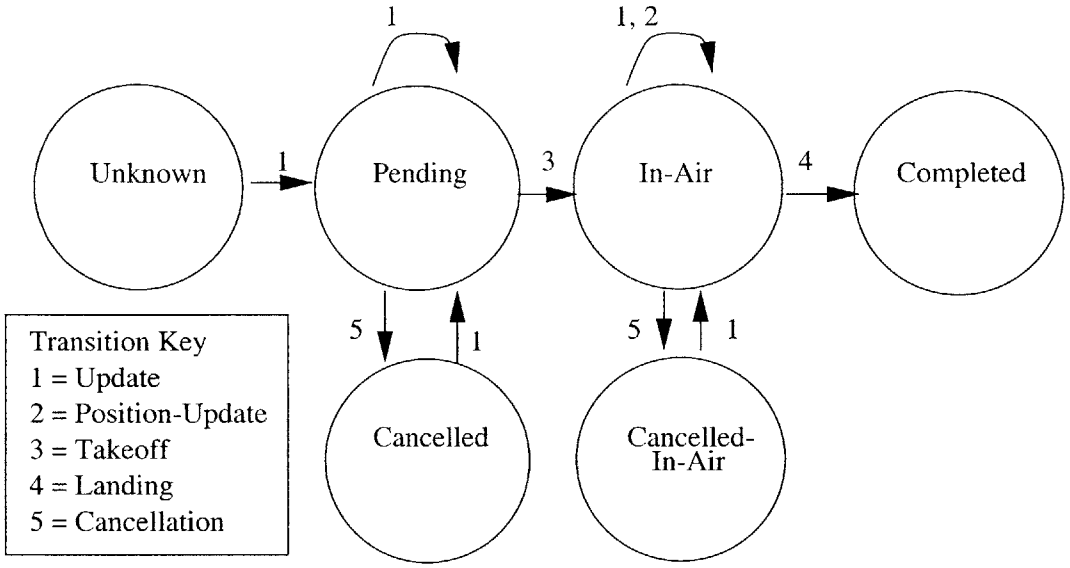
The typical flight history described above only accounts for 9 of the 21 listed transitions. The other 12 transitions allow for special cases that arise within the feed. The ActiveAir module was built to compensate for missing or ambiguous data that may occur during normal operation. These additional transitions are allowed to occur since it is possible to fill in any gaps that are caused by the ambiguity. The conjectured cause of these transitions and the method that they are dealt with are discussed in turn.

Unknown, Takeoff, In-air. This transition is caused by a missing flight plan (FZ) message that announces the flight before takeoff. A small number of flights are not announced before takeoff yet are fully tracked afterwards. It is possible that an error prevented the FZ from being sent over the feed, or not enough time was

Table 5.3: Finite State Machine Transitions

Initial State	Transition	New State
Unknown	Update	Pending
Unknown	Takeoff	In-air
Unknown	Position-update	In-air
Pending	Update	Pending
Pending	Cancellation	Cancelled
Pending	Takeoff	In-air
Pending	Position-update	In-air
Pending	Landing	Completed
Cancelled	Update	Pending
Cancelled	Cancellation	Cancelled
Cancelled	Takeoff	In-air
Cancelled	Position-update	In-air
Cancelled	Landing	Completed
Cancelled-in-air	Update	In-air
Cancelled-in-air	Position-update	In-air
Cancelled-in-air	Cancellation	Cancelled-in-air
Cancelled-in-air	Landing	Completed
In-air	Cancellation	Cancelled-in-air
In-air	Update	In-air
In-air	Position-update	In-air
In-air	Landing	Completed

Figure 5-2: "Normal" flight transitions



available before departure to send one out. Messages that do not receive a flight plan before takeoff are flagged as such in the database. (See section 5.4.8) A later FZ or boundary crossing (UZ) message is used to fill in the missing flight plan fields.

Unknown, Position-update, In-air. Similar to the previous case, both flight plan (FZ) and departure (DZ) messages were not received. Although flights entering US and Canadian airspace are usually announced with a DZ (which announces their airspace entry time, not their departure), often times they are not and position updates become the first messages received. Note that a new flight record will be created and put into the unknown state upon the arrival of boundary crossing (UZ) messages, with position update (TZ) messages simply being ignored. (See section 5.4.5) The missing flight plan information is filled in with the UZ message. The missing departure time is recorded as the time of the first received message instead, and is flagged in the database as such.

Pending, Position-update, In-air. Here, a departure message was missing despite the arrival of a flight plan (FZ) proposal. The departure message may not have arrived over the feed due to an error, or the lack of a departure in US or Canadian

airspace in the case of an international flight. The departure time is recorded as the time given in the prior flight plan proposal, and the time is flagged in the database as an estimate.

Pending, Landing, Completed. A number of flights seem to skip right to their landing after their initial flight plan proposal. These flights tend to be short in duration and it is possible that messages are not sent out during the span of their flight. Again, the missing departure time is filled in with the estimated time from the flight plan. Additionally, in-air flight statistics such as average and high speeds and altitudes are not recorded due to the lack of data.

Cancelled, Cancellation, Cancelled; Cancelled-in-air, Cancellation, Cancelled-in-air. Both of these transitions are the result of duplicate cancellation messages. Often times, flights traveling between US and Canada receive a cancellation message from authorities in both countries. No data is lost through this transition, and the plane simply remains in the same state that it was in previous to the message.

Cancelled, Take-off, In-air. This transition is caused by a missing flight plan message that reinstates the flight. In this case, tracking occurs as normal, ignoring the fact that the cancellation message was received.

Cancelled, Position-update, In-air. Here, a flight plan message and a departure message both were not received. The departure time given in the original flight plan is used, and again the previous cancellation is ignored.

Cancelled, Landing, Completed. In this case, all messages from the time of cancellation until arrival were not received. The flight record is built from the original flight plan and the arrival message.

Cancelled-in-air, Position-update, In-air. The flight was never reinstated after cancellation. Flight tracking occurs as normal, ignoring the previous cancellation.

Cancelled-in-air, Landing, Completed. No new messages were sent out from the time of the flight's cancellation until its arrival. The flight record is built with the information that was available, prior to cancellation.

5.4.3 Illegal Transitions

Although the finite state machine is flexible in getting around errors, a number of illegal transitions are still possible. Upon the occurrence of an illegal transition, an error message is written to the error log. One illegal transition represents a special case since it occurs on a regular basis. It is not written to the error log, but is instead handled to reflect an expected behavior:

In-air, Take-off. This message results from a duplicate departure message. Flights that travel between the US and Canada are tracked by authorities in both countries, who both send out a departure message. The second departure message signals the time of flight activation and not the actual departure time. To deal with this problem, the finite state machine does not allow departure transitions once the flight is in the air, resulting in the second departure message being ignored.

5.4.4 Flight Resolution Algorithms

The first challenge in constructing flight records out of individual messages comes in determining the flight corresponding to a message. At first, this may appear simple, since flights have an aircraft identifier with each message. Unfortunately, an aircraft identifier alone is not sufficient for resolving all messages. The first difficulty arises with multiple legged flights. Airlines commonly use the same flight number for flights that make a stop over in an intermediary city. Since flights are announced in advance of takeoff with flight plan announcements, a future flight leg can be tracked within the system at the same time as a current flight leg. A similar problem can be found with general aviation flights, in cases where an aircraft is making multiple flights in a short time span. Again, the system needs the ability to track multiple flights which all share the same aircraft identifier.

To solve this problem, flight resolution uses aircraft identifier, origin, and destination to form a unique identifier. Multiple legged flights plans are immediately differentiated with the use of this key. Note that the unique identifier differs from the unique key used in the database, which also includes the departure time. This is

because the differing temporal contexts of the database and ActiveAir module. With a time horizon of only a few hours, the ActiveAir table keeps track of flights within a relatively small time horizon. Unlike the database, flights with the same aircraft id, origin and destination are not repeated in this smaller time frame, making these three fields sufficient for matching messages.

Although aircraft id, origin, and destination give a means of resolving most flight messages, it cannot be used to match TZ messages, representing position updates. The most frequent message sent out over the ASDI feed, TZ messages do not contain the origin and destination within its body. The first observation that helps get around this obstacle is that TZ messages can only be sent out for planes that are in the air. If a flight has multiple legs under the same aircraft identifier, it is impossible for more than one leg to be in the air at a time. Thus, TZ messages can be resolved by matching the aircraft identifier in a message with the corresponding flight record that is currently in the air.

Unfortunately, a number of problems make resolution of TZ messages more complicated. The problems arise from the non-uniqueness of in-air flights having the same flight number. Although it is physically impossible for multiple legs of a flight to be in the air at once, errors in the ASDI feed and reporting do allow for this occurrence. Many flights are not transmitted with a full flight number. For example, some airlines transmit all flight identifiers over the feed using only the last three digits of the four digit flight number. Quite frequently, two flights sharing the last three digits of their flight number may be in the air at the same time. Thus, TZ messages for both flights would be sent out over the feed using the identical aircraft identifier. Another source of multiple active flights with the same identifier comes from missing messages in the feed. It is not uncommon for messages to be left out from the data stream. Particularly, a missing arrival message would incorrectly leave a completed flight in the air. If that same flight then departs to a new destination, more than one plane with the same flight identifier will be in the air at once. In this case, the TZ message should be associated with the later flight, since the older flight is no longer valid and is simply waiting for garbage collection to remove the record. (See section 5.4.6) Note

that these problems are all be solved with the use of the origin and destination, so they are not an issue for any messages except TZ's which require special treatment for these cases.

To help resolve TZ messages, the algorithm makes use of the computer identifier that is transmitted with the aircraft id. The computer identifier field is a device that allows the FAA systems to identify aircraft messages without the numerous difficulties that were encountered by the archival system. It would seem appropriate to just use the computer identifier from the beginning for all messages, avoiding the algorithms described here. Unfortunately, the utility it brings to the originating systems is not mirrored in the interpretation of the ASDI feed due to a number of inconsistencies. First, the computer identifier is not transmitted in all message types. Arrival (AZ) and boundary crossing (UZ) messages in particular do not transmit the computer identifier, necessitating the origin and destination for matching. A second problem arises from computer identifier disparity between different facilities. The ASDI feed combines data from a large number of monitoring facilities in both Canada and the United States. Some facilities do not transmit the computer identifier in their messages, while other times, different facilities may sometimes send out different computer identifiers for the same flight. Although it is not a perfect key that should be relied upon as the primary source of matching, the computer identifier is still useful as a backup. Since it is usually included in TZ messages, it serves as a good backup for matching these messages when there are multiple in-air flights with the same aircraft id.

Note that the algorithm assumes that a flight record already exists for the message in question. The process of creating new flight records is discussed in section 5.4.5. The full algorithm is summarized as follows:

```
if message is not TZ
  match message with flight having same aircraft id, origin and destination
else (if message is a TZ)
  if only one in-air flight with same aircraft id
    match message with in-air flight with same aircraft id
```

else (if more than one in-air flight with same aircraft id)

if message contains computer identifier

if only one in-air flight with same aircraft identifier and computer identifier

match message with in-air flight with same computer identifier and aircraft id

else (if more than one in-air flight with same aircraft identifier and computer identifier)

match message with latest in-air flight with same computer identifier and aircraft id

else (if message does not contain computer identifier)

match message with latest in-air flight with same aircraft id

5.4.5 Flight Creation

Obviously, not all messages can be matched to preexisting flights. Each flight must have a first message that begins its record. In general, a new flight record is created when certain messages are unable to be matched with any previous records. Particularly, a new flight record is created upon receiving departure (DZ), flight plan (FZ), and boundary crossing (UZ) messages. Limiting flight record creation to these particular messages determines the nature of the records inserted into the database. A balance had to be found between maximizing the number of flights recorded and the quality of the data written.

For a vast majority of flights, the first message received is the flight plan announcement (FZ). Thus, it is clearly necessary to create new flight records upon the arrival of these FZ message. However, restricting flight record creation to just FZ messages can result in the loss of meaningful information. Particularly, creating a new flight upon receipt of a takeoff (DZ) or boundary-crossing (UZ) message produces a flight record based on a partial history of the flight that is meaningful enough to store in the database. Although it is possible to create a new flight record for every message type, doing so would result in many meaningless records. For example, sometimes the only messages received about a flight are position updates (TZ). A flight record based on the information contained in TZ messages alone would be practically empty,

serving little value in the database. Similarly, creating a new flight record based on an arrival (AZ) message would produce a sparse record since the AZ message itself contains minimal data and no future messages will be received for a completed flight. In both cases, it is better to ignore the message, only creating new records upon more informative messages that come earlier in the life-cycle of the flight. A different line of reasoning motivated the decision not to produce new flight records for cancellation (RZ) and amendment (AF) messages. Both types of messages only have meaning in the context of previously received messages. An AF message is sent to change previously received data. Such a message simply does not make sense unless a preexisting flight record already exists. Similarly, an RZ message simply voids a previous flight. If no flight record exits, there is no flight record for the message to cancel and there is no point in creating a new flight record.

5.4.6 Flight Deletion and Garbage Collection

The first and most common method in which a flight record is removed form memory occurs when the flight is completed. The only message that completes a flight is an arrival (AZ) message. Whenever an AZ message is received, the corresponding record is updated with the contents of the message, and then removed from the flight table of tracked flights and stored elsewhere until the ASDISaver module transfers it to the database.

The second method in which flights can be deleted is through a garbage collection process that removes stale flights from the flight table of tracked flights. This process should not be confused with the standard Java garbage collection process although both have the same purpose, to clear memory that would otherwise be wasted. Often times, due to the unpredictability of the ASDI feed, a flight record gets created and is never completed. This can be due to an error where flight messages fail to be sent over the feed or a more allowable case of flights traveling outside US and Canadian airspace where contact is lost with the plane. For whatever reason, the system is left with a flight record in memory for which no new messages are sent.

The stale flights create a problem in two respects. First, these stale flight records

take up memory resources. Given enough time, the stale flights would pile up, exhausting the memory of the system, causing it to fail. Second, the system would be unable to properly deal with a new flight possessing the same aircraft id, origin and destination. This would actually be a frequent problem since airlines commonly run the same flight on a routine basis. Without removing such stale flights, a message for a new flight may be incorrectly matched to an old flight record that had been left around in memory.

To solve the problem, the ActiveAir module includes a separate thread that checks the age of the currently active flights and deletes those that are determined to be too old. A flight record is old when too much time has elapsed since the arrival of the last message for that flight. The thread itself is assigned a lower priority than the main thread that processes the ASDI feed and enters the new data into flight records. Thus, garbage collection occurs during periods of time in which no new data is sent through the feed. Even during peak traffic, ample time is still available for the garbage collection thread to run in between messages. The garbage collector simply iterates through each of the flights stored in the table of tracked flights, checking the time of the last received messages.

Flights are considered to be stale at different times depending on its flight state. This is because each state implies a different expected frequency of arriving messages. The amounts of time until a flight is considered old is given in table 5.4. Flights that are pending are allowed a large amount of time since it is possible for flights to be announced up to 12 hours in advance. [1, 21] From the time of its initial proposal, it is possible that no other messages will be received about the flight until it takes off, so it is important to give the flight record ample time before deletion. Once a plane takes off, however, position update (TZ) messages are sent out every one to five minutes. (ASDI Functional Description 19) Thus, a flight record for an in-air plane can be considered old in much less time when no new messages are received. Flights that are cancelled also allow an hour of time before removal. Sometimes, the ASDI feed contains cancellation (RZ) messages immediately followed by flight plan (FZ) messages for the same flight. Such a sequence is sent out as an alternative to

Table 5.4: Flight State Expiration Times

State	Time Until Old
Unknown	10 minutes
Pending	16 hours
Cancelled	1 hour
Cancelled In Air	1 hour
In Air	1 hour
Completed	10 minutes

amendment (AF) messages for changing flight information. The one-hour window allows cancelled flights the opportunity to be reinstated in this manner. The shortest amount of time is allotted for flights in the completed and unknown states. Flights in these states should never actually be in the active flight table, since completed flights are removed immediately after landing as previously described and the unknown state is used only when initializing the state of new flight records.

Once a flight record is determined to be stale, it is either deleted outright, or it is inserted into the database. Only the in-air and cancelled-in-air flights that are garbage collected are inserted into the database. In-air flights may have flown outside of US and Canadian airspace and are simply not being tracked anymore. Alternatively, it is possible that such flights landed and no arrival (AZ) message was sent out for it over the ASDI feed. For the cancelled-in-air flights, the aircraft may have simply been removed from the flight tracking system after takeoff. Enough information is known about such a flight to warrant its insertion into the database. At the time of removal, the arrival time is estimated to be the time of the last received message, and the arrival time validation flag variable is set to "database estimate", which indicates that the arrival time is not genuine but is the time at which contact was lost. (See section 5.4.8) Flights in all other states are deleted from memory, since they do not represent a valid flight. Stale pending flights are assumed to have never occurred and are deleted. Similarly, cancelled flights were never carried out and are deleted as well.

5.4.7 Data Structures

Two important data structures are used to store and manipulate the flight records, a table of tracked flights, and a queue of finished flights. Newly created flight records are originally stored in the table of tracked flights where it then undergoes the appropriate updates as additional messages are received. Once a flight is completed or garbage collected, the flight record is moved to the queue of finished flights until it is entered into the database.

Tracked flights are held in a hash table that uses the flight identifier as a key. Each key maps to a Java Vector object containing the flight records of all tracked flights with the same flight identifier. Since each key may map to more than one flight record, it is often necessary to provide additional parameters such as origin and destination, computer identifier, and flight state. A linear search is carried out for all flight records in the Vector to find which ones meet the additional qualifiers. Usually, there will be only one flight record for a given flight identifier. In the less frequent case, there should only be two or three records matching the key, keeping the linear search extremely small. The data structure was designed to efficiently carry out the flight resolution algorithms (see section 5.4.4) that require multiple keys for the different modes of access. The data structure takes advantage of the fact that there will never be a large number of flight records with the same flight identifier, allowing a negligible linear search rather then creating a more complicated multiple key hash table.

The table of tracked flights maintains the combination of flight identifier, origin and destination as a unique key. Upon every insert and update to the origin and destination fields of a flight, the records of all flights with the same identifier are checked to ensure that there is only one with the given origin and destination, throwing an exception if this is not the case. Thus, the data structure ensures that only one record can be returned whenever a flight is retrieved in the flight resolution algorithm based upon these three parameters. In contrast, it is possible for more than one record to have the same flight identifier, computer identifier pair. Thus, uniqueness is not

guaranteed by the data structure in this case, and the flight resolution algorithm chooses the matching flight for which a message was last received, as consistent with the flight matching algorithm.

The second important data structure used is the queue of finished flights. A finished flight is stored in this data structure until the ASDISaver module removes it and sends it into the database. The queue was designed to distinguish between two levels of priority, flights that were completed (received an arrival message), and flights that were garbage collected. A Java Vector object is maintained for each level of priority, and flights records enter and exit using a first-in first-out ordering. The data structure gives multiple modes of access, allowing for the removal of a flight with a specific priority level. Thus, an outside module, such as the ASDISaver can decide on its own database transfer policy. For example, the ASDISaver's policy of always removing completed flights before removing a garbage-collected flight is easily implemented using this data structure. (See section 5.5.1)

5.4.8 Updating the Flight Records

Flight records are constantly updated with the information received in ASDI messages. A description of the data stored by the flight record data structure is given in table 5.5.

Most of the fields of a flight record are straight forward, closely reflecting the data fields outlined in the ASDI message specification. Note that the record also holds information about the flights current and previous position, altitude and speeds. Although this data is not stored in the database, the ActiveAir module was designed as a reusable module that can be used in real-time applications as well as historical ones. The variables firstTime, lastTime, timeAltitude, timeSpeed, avgAltValid, and avgSpeedValid are used for the computation of average speed and altitude and are discussed separately in section 5.4.9. Each flight record also contains the aircraft's current state as determined by the finite state machine.

Table 5.5: Flight Record Fields

Field	Type	Description
aircraftId	String	The aircraft identifier
cid	String	The computer identifier
depPoint	String	The departure point
originalDest	String	The original destination
currDest	String	The current destination
aircraftType	String	The type of aircraft
heavy	String	The heavy indicator
numberAircraft	int	Number of planes in flight
equipQual	String	The equipment qualifier
assignedSpeed	String	The assigned speed
assignedAlt	String	The assigned altitude
route	String	The route
currAlt	String	The current altitude
currPos	String	The current position
highAlt	String	The highest altitude
highSpeed	String	The high speed
lastAlt	String	The previously recorded altitude
lastPos	String	The previously recorded position
currSpeed	String	The current speed
lastSpeed	String	The previously recorded speed
firstTime	Date	The timestamp of the first position update
lastTime	Date	The timestamp of the position update last received
timeAlttiude	float	The timealtitude used to compute average
timeSpeed	float	The timespeed used to compute average
etaFromDep	Date	The estimated time of arrival calculated at departure
arrTime	Date	The time of arrival
firstDepTime	Date	The first proposed departure time
actualDepTime	Date	The actual departure time
avgAltValid	boolean	Validates the average altitude
avgSpeedValid	boolean	Validates the average speed
departValid	int	Describes departure time
arriveValid	int	Describes arriveValid
proposedFirst	int	Whether proposed flight plan received before takeoff
planeState	FlightState	State of flight

Departure, Arrival and Flight Plan Descriptors

Each flight record contains three special fields, departValid, arriveValid, and proposedFirst that describe the other data fields, giving information about its origins. As mentioned, the ASDI feed does not always behave as expected and messages may not arrive that signal a change in the flight state. These three variable help describe the messages that were received, specifying the source and accuracy of other data fields.

The proposedFirst variable keeps track of whether or not a proposed flight plan was received before takeoff. The route, aircraft type, heavy indicator, equipment qualifier, assigned speed and assigned altitude are all sent within a flight plan. Usually, this information is transmitted and stored before departure. Although this information is sometimes repeated after takeoff, the route field is often not the same, starting from the airplane's current location, instead of the origin airport. Additionally, the assigned speed and altitude are replaced by the airplane's requested speed and altitude. Thus, there are elements of the flight plan that can only be attained in a message before departure. Unfortunately, not all flights in the ASDI feed are announced with their flight plan before departure. A good number are first tracked starting with the takeoff, and some are tracked starting mid-flight. Rather then ignore the post-takeoff flight plans of these flights, the proposedFirst variable flag is set to zero (from one) to indicate that some of the fields did not originate from a proposed flight plan, and that the route, assigned altitude and assigned speed may convey slightly different information. (The variable is saved as an integer rather than a boolean value for compatibility with the Oracle database which does not have boolean variables.)

The departValid and arriveValid tags describe the actual departure time and arrival time fields, respectively. The two fields describe the source of their respective times, allowing four integer values corresponding to valid (0), asdi-estimate (1), db-estimate (2), and cancelled-estimate (3). Valid indicates that the time is the actual departure/arrival time, sent through a departure (DZ) or arrival (AZ) message, respectively. ASDI-estimate indicates that the time is an estimated arrival or departure

time sent out over the ASDI feed. Both DZ and AZ messages can qualify their included time as actual, or as an estimate. In addition, this value is used for flights for which a DZ message was not received at all, which instead uses the estimated departure time sent in the body of the proposed flight plan. The third value, db-estimate, is used when the ASDI feed fails to give any time whatsoever. For departure times, this occurs when no flight plan or departure message is sent out prior to the plane being in the air. In this case, the departure time is saved as the time of the first received in-air message, providing the latest time that the plane could have departed. For arrival times, this value is used when no AZ message is received for an in-air flight, and the garbage collector ends up removing it. In this case, the arrival time is saved as the time of the last received message, providing the earliest time that the plane could have landed. Finally, the arriveValid variable allows a fourth value, cancelled-estimate. Like db-estimate, this value is used for flights for which messages stopped arriving mid-flight. However, this value further qualifies the flight as being cancelled mid-flight. Again, the garbage collector fills in the arrival time with the time stamp of the last received message.

Updating Contexts

When updating the fields of each flight record, the ActiveAir module uses both the message type and plane state for contextual interpretation. It is possible for a message field to appear in more than one type of message. The same data field may also be valid for multiple states of the plane, offering different meaning when it is in each one. It is important that different behaviors be carried out, depending upon the context of the message. The first context used is the message type. A different method handles the updating of each of the different types of messages. An alternative would have been to use one standard method that handled ever message types, producing different behaviors on the field level. However, doing so would produce a uniform behavior that is not always appropriate for all types of messages. For example, amendment (AF) messages use the same fields as a flight plan (FZ) message to describe the flight identifier. However, an amendment containing a change to a flight identifier must

also change the hash key in the table of tracked flights, while a new flight plan simply needs to produce a new flight. Another important use of the message content is in the proper handling of flight state. Many of the message types have an effect on the state of the plane, causing it to change state, regardless of the individual fields contained within. The arrival of a cancellation (RZ), departure (DZ) and arrival (AZ) messages changes the state of the plane besides updating the fields. The second context used in updating flight records is the plane state. The plane state is used for two important purposes, to provide further context for messages that can occur at any time in the flights history, and to provide a mode of error checking. Some messages such as amendments, cancellations and flight plans can occur prior to takeoff as well as in the air. Some fields need to be handled differently depending on the state of the plane. For example, the route, which can be transmitted in both FZ and AF messages is only updated prior to takeoff since once the plane is in the air the route starts from the plane's current position rather than the originating airport. In its error checking mode, the plane state helps detect the occurrence of duplicate messages. For the case of duplicate departure messages, the plane state determines that the second message, which provides less accurate information (see section 5.4.3), should be ignored. By checking the plane state, it is possible to learn that a departure was already recorded for the flight and the record should not be erroneously updated to reflect the information in the second departure message.

5.4.9 ASDI Field Processing

Many of the data fields transmitted over the ASDI feed require additional computation before they are stored in the database. Some fields simply require additional parsing to break the it into separate components. A few fields need more complicated processing, however, requiring both calculation and the addition of contextual information not sent over the feed.

Time Processing

Since it describes air traffic in real time, the ASDI feed sends all departure and arrival fields with hour and minutes information only in UTC time. Clearly, a historical database would need more detailed information, requiring a full date to distinguish between flights. Thus, the ActiveAir module augments each time field with the date, based upon the current date and time. However, it is not correct to simply attach the current system date (after adjusting to UTC time) onto each time that passes through the feed. Although in most cases a time coming through the feed refers to the current day, this is not the case for times near 00:00 UTC time. A proposed departure time may be sent out in the closing hour of one day, announcing the departure after 00:00 UTC on the next day. Similarly, actual departure and arrival times may refer to events from the previous days in the opening hour after 00:00 UTC. Thus, it is necessary to use a more complicated algorithm to assign dates, which can be accomplished by assigning the closest date to the time in question. First, the algorithm hypothetically assigns the time to the current day, the previous day and the next day. It then takes the difference between each of the three times and the current time, accepting the date that generates the smallest absolute difference. For example, at 23:00 UTC on May 15 2000, a message is sent over the feed announcing a 1:00 UTC arrival. The algorithm would try 1:00 UTC on May 14, 15 and 16. It will take the absolute difference between each of these and the current time of 23:00 UTC, discovering a difference of 46 hours for May 14, 22 hours for May 15, and 2 hours for May 16. Thus, it will assign May 16, 2000 1:00 UTC as the date. One item to note is that the algorithm always examines all three dates, even though it would seem that proposed times only refer to the current or future day and actual times only refer to the current or past day. However, this is not always the case and proposed times may sometimes refer to times that have already passed and actual times may refer to times in the future. Lag time in the transmission of records and a lack of synchronization between computers in the air traffic network may result in these unexpected occurrences. Fortunately, by checking all three dates, the "best" time is assured to be assigned.

Speed and Altitude Processing

The database stores the average and high speeds and altitudes recorded by each plane. These four pieces of data are not transmitted over the feed but are in fact calculated from the individual speed and altitude fields that are transmitted in the position update (TZ) messages. The calculation of the high speed is trivial with the ActiveAir module simply recording the highest speed and altitude that appears in any TZ message. The calculation of average speed and altitude is more complicated. Although the following description refers to speed, both use the same method of computation.

The average speed stored by the system is computed over the time period beginning with the first TZ for the flight, and ending with the last TZ. For every pair of consecutive TZ messages, a simple average is taken between the two speeds supplied. This average speed is then multiplied by the amount of time separating the two TZ messages, giving a speed-time product. This product is then added to a running sum that includes the products of every previous consecutive pair of TZ messages. At the end of the flight, this sum is divided by the total time, giving the average speed for the flight. The resulting average is time weighted, compensating for non-uniformity in the inter-arrival times of TZ messages.

This method of computing the flight's average speed contains a few sources of error. First, the averages do not take into consideration the time between takeoff and the first TZ message, and the time between the last TZ message and landing. Fortunately, these time periods are expected to be small and would not have a large effect on the resulting averages. Second, the times associated with each speed and altitude are non-exact. The calculation uses the time stamp sent with the TZ message that represents the time the message was generated. Thus, a delay exists between the recorded speed and altitude and the time the message is generated by an FAA facility. Fortunately, this delay is expected to be small and any changes occurring within this time period would be insignificant. A third source of error is the assumption that the plane undergoes constant acceleration and change in elevation between any pair

of consecutive TZ messages. If this assumption was grossly untrue than the simple average of the speeds or altitude would not reflect the planes true position and speed in the corresponding time period.

Although the altitude and speeds transmitted in the TZ messages are in valid formats, there are times when the values are missing or are not helpful in the calculation. For example, an altitude may contain an "on-top clearance" designator which specifies that an exact altitude is not given and the plane can be anywhere above the given altitude. In cases where data is missing from the messages or is unusable, the calculation is not performed and the avgAltitudeValid (or avgSpeedValid) boolean flag is set to false. When writing the data to the database, a false value indicates that the average calculation was not possible and the field should be left blank in the database. Some of the other designators that are sent over the feed do not have meaning that affect the calculation and they can be ignored. For example, a C following the altitude specifies that the plane is out of the range it was supposed to be in. Although interesting, this designator has no affect on the actual average altitude. A special case is the transmission of block altitudes, in which the plane can be between two given altitudes. In this case, high altitudes are recorded using the maximum altitude in the block and average altitude calculations use the average of the two block boundaries.

Airport name standardization

Another area requiring special processing is the origin and destination airport identifiers. Unfortunately, different transmitting facilities may use different identifiers for the same airport. The origin and destination of a single flight can be referred to differently in messages sent out by different facilities. This problem usually arises with flights traveling between the United States and Canada. Since the origin and destination are used as keys to match messages to flight records, (see section 5.4.4) it is important that the ActiveAir module convert all airport identifiers to a single standard. Doing so also improves the data stored in the database since queries would only need to refer to the standard identifier rather than trying all possibilities for

an airport. Clearly, some function is necessary to convert all airport identifiers to a standard.

Fortunately, it is possible to standardize the airport identifiers without having to store a full listing of the different identifiers for all the world's airports. Airport identifiers are transmitted either in a three-letter IATA format, or a four-letter ICAO format. All airport identifiers outside the United States and Canada are transmitted with the four letter ICAO codes. The problem arises in the transmission of flights inside the US and Canada. Canadian facilities transmit these flights using the four letter ICAO code while US facilities use the three letter code. Fortunately, IATA and ICAO codes for US and Canadian airports have the property that their IATA code is the ICAO code minus the first letter. For example, Logan Airport in Boston is identified as KBOS in ICAO and BOS in IATA. Note that this property does not hold in general, airports outside the US and Canada do not have corresponding codes. Furthermore, all US and Canadian airport codes are easily distinguished from other ICAO codes since all US airports begin with a "K" and all Canadian airports begin with a "C".

In the database, all Canadian and US airports are referred to by their three letter IATA code. Any four-letter ICAO identifier beginning with a "C" or "K" is converted to its corresponding IATA code by simply removing its first letter. Since all other airports are always transmitted in the same format, they can be left alone. Thus, standard formatting can be attained without having to store any additional knowledge about code conversion in memory, referring to all US and Canadian airports by its IATA and all other airports with its ICAO designator.

5.5 ASDISaver

The ASDISaver module is the center of the archival system. This module handles all communication and interaction with the other three modules, retrieving parsed data from the feed, sending it to the ActiveAir module for tracking and flight record formation, before finally sending finished flight records to the database. Since the

module interacts with a continuously running feed, it is important that it carry out its job efficiently, performing its various tasks as necessary in concurrent operation. As the core of the working system, the ASDISaver module provides ample points of failure, which require it to be robust when handling the loss of any of the other modules.

5.5.1 Interaction with ActiveAir module

The ASDISaver runs an ActiveAir module to handle the process of converting event messages to full flight records. Every message retrieved by the ASDISaver is immediately passed to the ActiveAir module for processing. After processing the new message, the ASDISaver module checks if the ActiveAir module has any flights that were finished and ready to be placed into the database. If finished flights exist, one record is removed from the ActiveAir module and sent into the database. The loop then processes the next message, always removing at most one flight for every message received.

The queue of finished flights made available by the ActiveAir module contains two types of finished flights. The first type is flights that are verifiably completed, meaning that an arrival (AZ) message was received for them. The second type is flights for which messages stopped coming and were removed through garbage collection. When removing flights from the queue, the ActiveAir module offers the ability to specify which type of flight to remove. Thus it is up to the ASDISaver to decide on a policy for removing flights.

As a rule, the ASDISaver attempts to enter a flight into the database as soon it is completed. This policy ensures that the database is as up to date as possible and minimizes the chance that a flight record is lost before it is entered. For flights receiving an AZ message, this policy would dictate that the flight be entered into database as soon as the message arrives. For timed-out flights, immediate entry does not seem as large as a priority since its actual completion time is uncertain. Still, these timed-out flights should not be allowed to wait around too long.

Thus, the ASDISaver achieves its desired behavior by always removing verifiably

completed flights from the queue before timed-out flights. Thus, every AZ message will result in the addition of a completed flight on the queue that gets immediately removed for entry by the ASDISaver. There will be at most one flight on the queue of completed flights, since they will always be removed immediately after it is entered. On the other hand, many timed-out flights can be on the queue at the same time. The ASDISaver removes these flight records after the arrival of any messages except AZ's. The queue of timed-out messages should never grow too long, however, since AZ messages only make up a small minority of the total messages. There are more than enough messages sent out over the feed to trigger the removal of all the queued flights, ensuring that they do not get stuck on the queue.

5.5.2 System Robustness

With many different pieces interacting together, it was important to build redundancy into the ASDISaver to prevent data loss. Because a system failure results in the loss of irreplaceable flight data, it is important that measures are taken to ensure the continuous operation of the system, providing a backup plan in case a piece of the system breaks down. As the focal point of the entire system, the design of the ASDISaver accounts for outages of the ASDI feed, database, and itself.

Database Outages

One possible failure point in the system would be the unavailability of the database. Since the database resides on a separate machine, the possibility exists for it to fail independently from the ASDISaver module. Although message collection and flight tracking can continue, the ASDISaver would be unable to store the finished flight records in the database as normal. As a solution, the ASDISaver was designed to store these finished flight records locally, until the time that the database becomes available again. Once the database comes back on-line, the saved flight records, as well as all new flights records, are sent to the database. The writing of the saved local flight records is carried out with the same methods used to save new flight records.

That way, if the database goes down again during the recovery efforts, the saved records would be "re-saved" locally instead of being lost.

The status of the database is determined by the results of each database write. An attempt is made to send every flight record to the database. The JDBC driver either gives confirmation of a successful transaction, or returns an error message. The error message can indicate the unavailability of the database, signaling to the ASDISaver that records should be stored locally. Similarly, a successful transaction signals to the ASDISaver that the database is running, prompting it to dump any saved local messages, and send future records directly to the database.

The ASDISaver module provides two ways of storing flight records, in the system memory and in a file stored locally by the system. The choice of storage is selected by the user upon startup of the ASDISaver along with a file name if this option is chosen. The flight records are stored as SQL statements using the same lines of code that would have placed the record into the database. Upon the reawakening of the database, the SQL statements are simply sent back to the database in the format in which they were stored.

Storing flight records in a file has a number of benefits over storage in memory. First, the file provides a more permanent record of the flight. Whereas a shutdown of the ASDISaver would result in the loss of all flights stored in memory, records stored in the file would still exist. As an extra safeguard against losing flights in memory, the file is flushed and saved with the addition of each new record to ensure that nothing is left in the memory buffer. Thus, file storage offers more protection against data loss. Second, an ASDISaver using file storage can continue to act as a backup much longer than a backup process using memory storage. Since Java does not swap out its memory, it is possible to exhaust the system's resources with flight records, causing the ASDISaver to breakdown. Saving records to a file allows more records to be saved, allowing the system to survive a database outage for a longer period of time. A third benefit is that the file affords more flexibility in data recovery. If desired, the file can be installed manually into the database, without the system having to recover the data on its own. It is easy to do this, since the file is made up

of SQL statements that can easily be executed in the database.

With these benefits, it would seem that the memory option would have no purpose. The option was included for two reasons. First, it is faster and more efficient to save messages to memory. This option would be preferable if redundancy was not considered a priority and lost data was not considered to be important. Second, the memory option allows easy portability between different systems. Whereas the file option requires the creation and storage of an operating system specific file, the memory option can be run on any platform without any further considerations.

ASDISaver Outages

The possibility of a failure of the ASDISaver also exists. When this occurs, the database may still be operational. However, flight data from the feed would have no way of being tracked and entered into the database. This problem has a simple solution, namely the operation of multiple ASDISaver feeds all connected to the same database. The running of multiple ASDISavers is aided by the implementation of the system in Java since it can be easily run on different platforms. By running the system on separate independent machines, a loss of any one ASDISaver would not stop the flow of information into the database. Since the ASDISaver instances all connect to the same feed, the data that each receives should be alike. Allowing multiple ASDISavers requires that the database have some mechanism to ensure that a flight is not entered twice. Additionally, by turning on the different ASDISavers at different times, it is possible for two ASDISavers to have different information about the same flight. Thus, the database must also be able to resolve cases where two ASDISavers attempt to write different information for the same flight. (See section 5.3.5)

ASDI Feed Outage

A third point of failure is the loss of the incoming ASDI feed data. Outages in the ASDI feed can be caused in many different ways, fitting into two general groups. One type is an error in the Volpe systems that send out and produce the feed. There is nothing that the ASDISaver can do about this type of failure since this is the only

source for the ASDI data. Fortunately, the Volpe center runs a robust server with its own redundancy to help ensure that data is always sent over the feed. A second type of failure occurs in the transmission between Volpe and the ASDISaver. This problem can be accounted for by setting up multiple pathways to the ASDI feed, running a separate ASDISaver on each line. Thus, if one ASDISaver loses the incoming data, the other ones would still be able to continue operation.

Once the feed is lost, the ASDISaver also puts in efforts to recover. After detecting the absence of data for a minute, the ASDISaver assumes that its data feed has died. This is a valid assumption since heartbeat (HB) messages are sent over the feed every ten seconds to provide a signal that the line is still running. Thus, the absence of data can never be interpreted as there being no airplanes in the air, however, unlikely that may seem. After determining that the feed is down, the ASDISaver attempts to restore its connection every ten seconds, attempting to open a new connection to the server each time. Thus, the ASDISaver is capable of picking up where it left off, using the data already stored in the ActiveAir table. When the data begins flowing again, the ASDISaver does not need to start from scratch. Hopefully, any data that was lost in the down time would be picked up by another running ASDISaver. The ability to recover is particularly beneficial in the case of brief outages where the loss of a small amount of data is not serious.

5.5.3 Handling Database Writes and Concurrency

One of the major design challenges posed by this module was ensuring the smooth and steady operation of processing incoming data and writing finished records all at once. Since the feed is continuously active, the module needs to ensure that database writes do not stall the processing of incoming data. Similarly, the module must also ensure that database writes do not pile up if too much time is given to processing incoming data. To alleviate this problem, the system creates a new thread for each completed database record. Each of these threads is in charged of writing the record to the database, dying upon completion. Creating new threads in this way helps alleviate the bottlenecks in the module. The delay from the write operation is due

to the time it takes for the record to travel to the database, the time it takes for the database to write the record to memory, and the time it takes for the confirmation message to be travel back over the network to the ASDISaver. Since none of these tasks require the use of the ASDISaver's processor, the thread responsible for writing the record is mostly inactive and is able to continue with message retrieval and flight tracking.

Note also that a new thread is created for each flight record, instead of just creating one thread to take care of all database writes. Although such a "writing thread" would be inactive most of the time, it would force records to pile up in a queue, sending records off one at a time. By performing each write in a separate thread, each record can be dispatched immediately, allowing the database to operate its own queue and work at its own pace.

As discussed previously, when the database is down, flight records are saved locally, either in memory or in a file. The same thread that attempts to write a flight record to the database is responsible for saving the record locally. Since multiple writer threads can be active at the same time, it is possible that they would attempt to access the file or data structure in memory concurrently, causing unexpected side effects. To ensure atomicity of the operation, the file and data structure are synchronized so only one thread can access it at a time.

A similar issue is encountered when recovering local messages and sending them to the database. While accessing the data structure in local memory or the file, it is possible that the database can go down again, requiring the ASDISaver to save a flight record at the same time that it is attempting to remove and recover an old message. An infinite loop would be created where a given flight record is removed from the front, only to immediately be placed at the end of data structure or file. To deal with this concurrency issue, the file or data structure is locked when the recovery process is begun. The name of the file is immediately changed, or the data structure is moved to a new reference. Now, the original file and the data structure using the original reference are empty. The lock can then be removed with restoration of the saved flight records commencing on the identical copies that were just created. If the

database does go down, the flight record would be saved in the empty file or data structure, without the danger of conflicting with the recovery process.

Chapter 6

Conclusion

The ASDI archival system that was built successfully meets all requirements initially outlined, providing a robust flight-based method of storing the ASDI feed. The system demonstrates a practical method of providing redundancy to a database writing process that can be applied to other systems that must continuously archive information. In converting the message based ASDI feed to historical records, valuable observations were made about the nature of the ASDI feed, providing insight into how it reports information about a flight over time. Indeed, the ASDI feed does not report each flight in the same way, allowing a flight's history to be transmitted through many different sequences of messages. Clearly, such observations would prove useful to other individuals interested in the behavior of the ASDI feed.

The system provides a useful tool for individuals interested in air traffic and aviation in general. The database offers numerous possibilities for applications that analyze and display the historical data. It is possible for the data to have an impact in the research of air traffic control and congestion, helping to alleviate the current problems facing the nation's transportation network.

Bibliography

[1] *Aircraft Situation Display To Industry Functional Description and Interfaces.* Volpe Center, Cambridge, Massachusetts, 3.0 edition, 5 March 1999.

[2] Michael A. Dornheim. Jump in delays shows atc bumping into capacity limits. *Aviation Week and Space Technology*, pages 90–94, 25 October 1999.

[3] Dimensions International Inc. Flightexplorer. http://www.flightexplorer.com/, 2000.

[4] The Trip.com Inc. Flighttracker. http://www.thetrip.com/, 2000.