NAME

mnesia_frag_hash − Defines mnesia_frag_hash callback behavior

DESCRIPTION

This module defines a callback behavior for user-defined hash functions of fragmented tables.

Which module that is selected to implement the *mnesia_frag_hash* behavior for a particular fragmented table is specified together with the other *frag_properties*. The *hash_module* defines the module name. The *hash_state* defines the initial hash state.

This module implements dynamic hashing, which is a kind of hashing that grows nicely when new fragments are added. It is well suited for scalable hash tables.

EXPORTS

init_state(Tab, State) -> NewState | abort(Reason)

Types:

```
Tab = atom()
State = term()
NewState = term()
Reason = term()
```

Starts when a fragmented table is created with the function *mnesia:create_table/2* or when a normal (unfragmented) table is converted to be a fragmented table with *mnesia:change_table_frag/2*.

Notice that the function *add_frag/2* is started one time for each of the other fragments (except number 1) as a part of the table creation procedure.

State is the initial value of the *hash_state frag_property*. *NewState* is stored as *hash_state* among the other *frag_properties*.

add_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)

Types:

```
State = term()
NewState = term()
IterFrags = [integer()]
AdditionalLockFrags = [integer()]
Reason = term()
```

To scale well, it is a good idea to ensure that the records are evenly distributed over all fragments, including the new one.

NewState is stored as *hash_state* among the other *frag_properties*.

As a part of the *add_frag* procedure, Mnesia iterates over all fragments corresponding to the *IterFrags* numbers and starts *key_to_frag_number(NewState,RecordKey)* for each record. If the new fragment differs from the old fragment, the record is moved to the new fragment.

As the *add_frag* procedure is a part of a schema transaction, Mnesia acquires write locks on the affected tables. That is, both the fragments corresponding to *IterFrags* and those corresponding to *AdditionalLockFrags*.

del_frag(State) -> {NewState, IterFrags, AdditionalLockFrags} | abort(Reason)

Types:

```
State = term()
NewState = term()
IterFrags = [integer()]
AdditionalLockFrags = [integer()]
Reason = term()
```

NewState is stored as *hash_state* among the other *frag_properties*.

As a part of the *del_frag* procedure, Mnesia iterates over all fragments corresponding to the *IterFrags* numbers and starts *key_to_frag_number(NewState,RecordKey)* for each record. If the new fragment differs from the old fragment, the record is moved to the new fragment.

Notice that all records in the last fragment must be moved to another fragment, as the entire fragment is deleted.

As the *del_frag* procedure is a part of a schema transaction, Mnesia acquires write locks on the affected tables. That is, both the fragments corresponding to *IterFrags* and those corresponding to *AdditionalLockFrags*.

key_to_frag_number(State, Key) -> FragNum | abort(Reason)

Types:

FragNum = integer()()
Reason = term()

Starts whenever Mnesia needs to determine which fragment a certain record belongs to. It is typically started at *read*, *write*, and *delete*.

match_spec_to_frag_numbers(State, MatchSpec) -> FragNums | abort(Reason)

Types:

MatcSpec = ets_select_match_spec()
FragNums = [FragNum]
FragNum = integer()
Reason = term()

This function is called whenever Mnesia needs to determine which fragments that need to be searched for a *MatchSpec*. It is typically called by *select* and *match_object*.

SEE ALSO

mnesia(3erl)